NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN ANALYSIS OF DISC CARVING TECHNIQUES

by

Nicholas Mikus

March 2005

Thesis Advisor:	Chris Eagle
Second Reader:	George Dinolt

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2005	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: An Analysis of Disc Carving Techniques			**5. FUNDING NUMBERS**
6. AUTHOR(S) Mikus, Nicholas			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Disc carving is an essential element of computer forensic analysis. However the high cost of commercial solutions coupled with the lack of availability of open source tools to perform disc analysis has become a hindrance to those performing analysis on UNIX computers. In addition even expensive commercial products offer only a fairly limited ability to "carve" for various files.

In this thesis, an open source tool known as Foremost is modified in such a way as to address the need for such a carving tool in a UNIX environment. An implementation of various heuristics for recognizing file formats will be demonstrated as well as the ability to provide some file system specific support.

As a result of these implementations a revision of Foremost will be provided that will be made available as an open source tool to aid analysts in their forensic investigations.

14. SUBJECT TERMS Computer Forensics, Disc Carving, Data Carving			**15. NUMBER OF PAGES** 159
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN ANALYSIS OF DISC CARVING TECHNIQUES

Nicholas A. Mikus
Civilian, Federal Cyber Corps
B.S., University of Illinois Chicago, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2005

Author: Nicholas Mikus

Approved by: Christopher S. Eagle
Thesis Advisor

George Dinolt
Second Reader

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Disc carving is an essential element of computer forensic analysis. However the high cost of commercial solutions coupled with the lack of availability of open source tools to perform disc analysis has become a hindrance to those performing analysis on UNIX computers. In addition even expensive commercial products offer only a fairly limited ability to "carve" for various files.

In this thesis, an open source tool known as Foremost is modified in such a way as to address the need for such a carving tool in a UNIX environment. An implementation of various heuristics for recognizing file formats will be demonstrated as well as the ability to provide some file system specific support.

As a result of these implementations a revision of Foremost will be provided that will be made available as an open source tool to aid analysts in their forensic investigations.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No.DUE-0114018.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

This paper, as well as most things in my life, would not have been possible without my wife Holly.

I would also like to thank Jesse Kornblum and Kris Kendall for developing the open source tool Foremost for analysts to use and learn from.

Finally I would like to thank LCDR Chris Eagle for teaching me to be “leet”.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

As computers become more prevalent in society, their use for criminal and other nefarious purposes also increases. This has lead to a demand for Computer Forensic specialists to analyze digital evidence to help catch these criminals. In response to this demand the FBI and other law enforcement agencies are building Regional Computer Forensic Laboratories across the country. These facilities are equipped with state of the art tools and highly trained examiners to help with an overwhelming case load. In FY 2003, the RCFL Program processed 82.3 terabytes of data; this is the equivalent of roughly 3,427,644 boxes of paper filled with text [Ref. 1]. The San Diego RCFL alone received over 700 requests to review various cases involving the need for computer forensic examinations. This shows the magnitude of the increase in the computer related evidence, and the bad news is, it is only going to get worse for examiners. As hard drives and multimedia storage devices grow exponentially so must the capabilities of the tools which investigators use to analyze these devices. One major area that must be improved is referred to as disc carving.

Disc carving is an essential aspect of Computer Forensics and is an area that has been somewhat neglected in the development of new forensic tools. The term disc carving can be defined as data recovering using "raw" information as opposed to file-system meta-data. Disc carving has a great impact on computer forensic cases because it adds the flexibility of being able to dissect stored information independent of any underlying file system structure. Disk carving has also become synonymous with the term data carving but for the context of this paper the term disc carving will be used. My research in the arena of disc carving will aid investigators in being able to extract useful information from storage devices using an open source product which can automate a large portion of the process. Making this tool and its source code freely available eliminates one of the greatest inhibitors which is the cost of many commercial forensic suites.

New approaches to disc carving must be studied to help develop more efficient and reliable products for investigators to use. These methods can hopefully offset some of the increasing work load that high volume storage devices pose to limited number of investigators. In addition this research can help in the prosecution of criminals who use computers in some form or another in the conduct of their business.

A. DISC CARVING BACKGROUND

Disc carving refers the ability to recover files from a medium which may or may not be a recognizable file system. It is commonly used in reference to extracting files from unallocated or slack space from a given file system [Ref. 2]. Files are allocated disk space in multiples of the file system block size. Slack space refers to the unused space within the last block allocated to a file. This space lies between the last data byte of the file and the end of its associated block. The amount of slack space a file contains can be computed as (file size) modulo (block size). Thus since all files do not end exactly on block boundaries this “excess” space can be used to hide data from file system view.

Disc carving research has been relegated to the background of forensic tool development. Tools such as ILOOK [Ref. 3], Encase [Ref. 4], and FTK (Forensic Tool Kit) [Ref. 5] focus on recovering files via metadata. It is true that this is a very effective and efficient method of file recovery, however, if the metadata is corrupted or non-existent, then these methods usually fail. Also the data in question could have been “deleted” from the file system view. However, the data could very well be, and often is still intact on the disc, it is just a matter of “carving” it out. In my experimental results data that is years old can often be recovered from unallocated space, depending on the volume size and disc activity.

FTK and Encase address the issues of data carving but these tools are Microsoft Windows based and are very expensive. The cost of these tools and the fact that the extraction methods are closed source is an inhibitor to the forensic community that wishes to use a more robust tool that can perform successful extractions. ILOOK is another Microsoft Windows based tool used in forensic investigations but it is only available to Law Enforcement and government agencies. ILOOK is free to specific government agencies that support a law enforcement directive; however, like FTK and Encase, it is closed source. Thus the ability to learn from and improve extraction methods is diminished. The fact that the majority of tools currently used by law enforcement are closed source has lead some developers and forensic researches to turn to the open source community.

In the open source world Brian Carrier's Sleuthkit has become a standard tool for doing forensic analysis on UNIX systems. This tool has provided a wealth of resources to examiners that use a UNIX platform and also those faced with fiscal constraints who cannot afford its Windows counterparts. However, one glaring hole in the Sleuthkit is that it provides no carving functionality. Thus investigators looked to a tool named *Foremost* to fill in the gap. Foremost is a very powerful disc carving tool but it is lacking in some respects as chapter II will discuss. The eventual inclusion of disc carving functionality in Sleuthkit will help solidify its place in the forensic community and provide a viable alternative to commercial products.

B. PURPOSE OF STUDY

The purpose of this research is to develop a more intelligent tool to extract files from a medium independent of its file-system structure. Such a tool will greatly reduce the time spent by investigators plowing through binary file representations trying to ascertain what files can and cannot be recovered. Current open source methods of disc carving lack the sophistication needed to provided a robust disc carving program. The general idea to develop such a tool is to mimic the behavior of the *file* command available on UNIX systems but to apply that intelligence to the disc carving tool Foremost. Foremost is a utility that "carves" files out of raw data blocks based on file header and footer data. The file command, which will be covered in depth in chapter II, often looks at more than just the header of the file in order to comprehend the file's internal data structures as well. If the functionality of file and Foremost were combined then a much more powerful tool could be produced. The strategy that emerged as the most fruitful in the development of extraction methods was to perform a more detailed analysis of specific file data structures, allowing for a more in depth recognition as well as increasing the speed of the program. Speed is obviously key when performing analysis of very large disc images, the data structure approach does require the program to become more intelligent but it will save time for the examiner who is currently required to at least have a working knowledge of file format specifications in order to successfully recover files manually. The automation of this process however challenging, offers great promise in terms of productivity.

My research produced many extraction algorithms which can then be scrutinized and tested via the vast open source forensic community. Creating open source forensic tools is a great way to develop and test tools economically and efficiently. The current implementation of the algorithms described in chapter III can be viewed in the CVS repository of Foremost at http://cvs.sourceforge.net/viewcvs.py/foremost/foremost-1.0/. The availability of the enhancement has lead to increased feedback from the forensic community about features they would like to see as well as problems they encounter.

The outcome of the cycle of publishing and revising the source code will eventually lead to a more robust library of extractions methods that can essentially do the "dirty work" of looking at blocks of data trying to determine if the file is still intact and what type of file is it. Tools like Foremost solve many problems but also introduce new ones. However, these problems may be viewed in a positive light because their solutions lead to more intelligent and efficient products that can aid analysts in data carving.

The debate against open source is usually that the software product may be more prone to exploitation. This is not a major concern with Forensic software as it is not providing a service to multiple clients, just analyzing a local drive. Thus in the case of forensic software, using open source tools just makes more sense.

The goal of a good disc carving tool is to remain file-system independent, which ensures the flexibility of being able to analyze a wider range of storage media. However, options should be added if knowledge of the file-system of a given device is obtained. One example of this is the problem that indirection blocks, used in UNIX file-systems, pose to disc carving. This issue is covered in great detail in chapter three and is another area that commercial forensic products fail to address in the context of disc carving. Thus this paper will describe the implementation of algorithms which will enhance extraction capabilities of an existing Forensic tool, independent of file-system structure, but also, when possible, leveraging certain file-system attributes that can aid the extraction process.

C. THESIS ORGANIZATIONON

This paper will present a working implementation of a disc carving tool that can recover specified files from any block of raw binary data such as, but not restricted to, partial or complete disk images. Chapter II details the operation of Foremost and the file command and explains how a hybrid will benefit the forensic community. Chapter III will provide a description of the important algorithms and the details of their construction. The algorithms include file extraction methods as well as indirection block detection for UNIX file-systems. Full source code examples of each extraction algorithm are provided in Appendix A. Chapter IV will provide a set of experimental results when running the foremost enhancement versus various data carving tools. Different files systems are discussed and tested as well as the details of the indirect block detection capabilities. Chapter V will conclude my research by discussing problems faced as well as describe future work in this are of Computer Forensics.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. FOREMOST

Foremost is an open source forensic tool created for the Linux platform and developed by Special Agents Kris Kendall and Jesse Kornblum of the U.S. Air Force Office of Special Investigations. In accordance with 17 USC 105, this tool is not afforded any copyright protection because it is a work of the U.S. government. The tool was inspired by, and designed to imitate the functionality of, the DOS program CarvThis, written by the Defense Computer Forensics Lab. Foremost enables forensic examiners to automatically recover files or partial files from a bit image (or the media itself) based on file header and footer types specified in a user-defined configuration file.

Foremost works by reading into memory a pre-defined portion of the media or media image under examination. By default this chunk of memory is 10MB, thus images are analyzed 10MB at a time. Each chunk is searched for file headers contained within the Foremost configuration file. If a matching header is found, then Foremost attempts to locate the corresponding end of the file. Foremost will search for the footer (which signifies the end of the file) until a file size limit listed in the configuration file is reached. If the footer is found then the recovered file data is written to a separate disk file, however if it is not then Foremost will dump the maximum file size after the header. If no footer is defined in the configuration file then foremost will extract the maximum number of bytes specified by the configuration file after every header is found. Using a file size limit serves as a means to stop Foremost from adding data to a recovered file if the appropriate file footer is not found. This is a fairly efficient approach if such a header/footer pair is uniquely defined but this is not often the case.

Another limitation of Foremost is the fact that even if a file is successfully extracted, the same data that was just analyzed is checked again. This method is designed to recover embedded files containing the header signature but can be very computationally expensive. This implementation is flawed in the case where Foremost cannot determine the end of the file, thus it merely dumps a predetermined amount of data, this data is then searched for the same header. Files that contain multiple headers result in fragments of files being written to disk often resulting in the creation of multiple garbage files. This reduces the speed of the program as time is wasted re-analyzing and re-extracting data that has already been extracted as part of a larger file. This added execution time could be better spent ensuring a valid extraction in the first place rather than relying on forensic specialists to wade through redundant fragments of a given file.

Table 1 illustrates some sample Foremost configuration file definitions. The first field denotes the suffix appended to the file if extracted, the second defines whether the search to be performed is case sensitive, followed by the maximum defined file size and lastly the header/footer pair. Notice the definition for *avi* doesn't include a footer; this is a common occurrence in the configuration file. If this is the case then Foremost will just extract the maximum amount following the header, often leading to truncated extractions. Other formats in the configuration file that do not contain an adequate footer include doc, mov, bmp, xls, java.

Suffix	**Case Sensitive**	**Max Size**	**Header**	**Footer**
jpg	Y	20000000	\xff\xd8\xff\xe0\x00\x10	\xff\xd9
htm	N	50000	<html	</html>
avi	Y	4000000	RIFF????AVI	

Table 1. Foremost configuration file

These formats show the flawed method by which these files are extracted. The program then relies on a forensics analyst to extract useful information from the maximum file amount. This amount may not be of sufficient size, thus forcing the analyst to increase the file size and re-run the program iteratively until enough of the file has been extracted. This is an added burden to the time consuming task of performing a detailed analysis of very large storage devices. If this process could be made more intelligent then examiners could spend more time analyzing the evidence rather than extracting it.

B. FILE

File is a program which examines a given file's content in an attempt to classify it based on the actual data in the file rather than merely the suffix (.exe) [Ref. 6]. There are three sets of tests that are performed by *file*: file system tests, magic number tests, and language tests. The first test that succeeds causes the file type to be printed. The idea of the Foremost enhancement is to harness the same type of built-in intelligence provided in the magic number tests.

The determined file type will usually fall into one of the following categories: *text* (the file contains only printable characters and a few common control characters and is probably safe to read on an ASCII terminal), *executable* (the file contains the result of compiling a program into a binary form understandable by some operating system), or *data* meaning anything else (data is usually `binary' or non-printable). Exceptions are well-known file formats (core dump files, tar archives, etc.) that are known to contain binary data. When modifying the /usr/share/magic file or the program itself, it is necessary to preserve these keywords. Note that the file /usr/share/magic is built mechanically from a large number of small files in the subdirectory Magdir in the source distribution of this program, these files can be modified by a user knowledgeable about a specific file specification.

The file system tests are based on examining the return from a stat(2) [Ref. 7] system call. The program checks to see if the file is empty, or if it's some sort of special file. Any known file types appropriate to the system you are running on (sockets, symbolic links, or named pipes (FIFOs) on those systems that implement them) are discovered if they are defined in the system header file *<sys/stat.h>*.

The magic number tests are used to check for files with data in particular fixed formats. The canonical example of this is a binary executable (compiled program) *a.out* file, whose format is defined in *a.out.h* and possibly *exec.h* in the standard include directory. These files have a `magic number' stored in a specific, well defined location near the beginning of the file that tells the UNIX operating system that the file is a binary executable, and which of several types thereof. The concept of `magic number' has been adopted by the developers of many other data file formats. Any file with some invariant identifier at a small fixed offset into the file can usually be described in this way. In the Linux operating system, the information identifying these files is read from the compiled magic file */usr/share/magic.mgc* , or */usr/share/magic* if the “compiled” file-magic.mgc does not exist. Notice Table 2 which shows how the standard JPEG header is defined in the magic file. More tests are performed to determine more information about the image but the principal of the program is that it looks at the data structures of the file as opposed to just header information.

Offset	**Data Type**	**Data to match**	**Description**
0	Beshort	0xffd8	JPEG image data
>6	String	JFIF	\b, JFIF standard

Table 2. FILE sample magic format

If a file does not match any of the entries in the magic file, it is examined to see if it seems to be a text file. ASCII, ISO-8859-x, non-ISO 8-bit extended-ASCII character sets (such as those used on Macintosh and IBM PC systems), UTF-8-encoded Unicode, UTF-16-encoded Unicode, and EBCDIC character sets can be distinguished by the different ranges and sequences of bytes that constitute printable text in each set. If a file passes any of these tests, its character set is reported. ASCII, ISO-8859-x, UTF-8, and extended-ASCII files are identified as ``text" because they will be mostly readable on nearly any terminal; UTF-16 and EBCDIC are only ``character data" because, while they contain text, it is text that will require translation before it can be read. In addition, *file* will attempt to determine other characteristics of text-type files. If the lines of a file are terminated by CR, CRLF, or NUL, instead of the Unix-standard LF, this will be reported. Files that contain embedded escape sequences or overstriking will also be identified.

Once the *file* program has determined the character set used in a text-type file, it will attempt to determine in what language the file is written. The language tests look for particular strings that can appear anywhere in the first few blocks of a file. For example, the keyword “.br” indicates that the file is most likely a troff(1) input file, just as the keyword struct indicates a C program. These tests are less reliable than the previous two groups, so they are performed last. The language test routines also test for some miscellany (such as tar(1) archives). Any file that cannot be identified as having been written in any of the character sets listed above is simply said to be ``data"[Ref 6.].

These tests and the ability to define new tests based on the file offsets prototype for the types of logic that must be incorporated into a program like Foremost to make it more effective. The only thing *file* lacks for our context is a looping structure. In addition it doesn't concern itself with embedded files or where the file data terminates. [1]However applying this functionality is relatively trivial once the data structures of the file are adequately understood. File specifications are the key to utilizing the searching capability that Foremost provides in the most efficient manner.

1 An embedded file refers to a FILE that is encapsulated within another file.

THIS PAGE INTENTIONALLY LEFT BLANK

III. IMPLEMENTATION

A. HEURISTICS

1. OLE Archive

Microsoft's Object Linking and Embedding file format provides for a "structured storage" environment for various types of file formats [Ref. 8]. It is basically an abstraction so that file formats can use the OLE API to read and write data to the disk. This is useful because the formats can then store the data as objects instead of a flat file. It also permits more cross functionality between applications that adhere to this file structure, therefore it is easier to copy objects from a Word document to an Excel file for instance. However this also significantly complicates file extraction because the file structure is much more dynamic.

Previously Foremost only provided the OLE header for Microsoft Word documents and extracted the following the first 50KB relying upon the examiner to determine the end of the file. The algorithms presented here provide a much higher rate of extraction with increased accuracy of the data recovered. These algorithms make use of an API developed by the Chicago Project (http://chicago.sourceforge.net/) whose goal is to develop a C library to read and write Microsoft Excel documents [Ref. 9]. This API was modified to add error detection and the ability to analyze an array of bytes as opposed to a stand alone file. This enables Foremost to use this API to extract file dependent information and determine what type of file was stored in an OLE structure.

Parsing the OLE data structures proved complicated but extremely rewarding because the extraction of any interesting Microsoft File Format adhering o the OLE format became trivial. The algorithm works by first reading the header block which is always 512 bytes. The block size of the remaining document is defined in the header but it is usually 512 bytes as well. This value is specified by the uSectorShift field located in the header block which is outlined in Table 3 below. This table also provides information about what data values are located within the OLE header and Table 4 provides a hexadecimal display of an OLE header taken from a Word Document. Table 4 also shows the magic number, uByteOrder, num_FAT_blocks, and the root_start_block in bold as these fields are crucial to begin parsing the OLE data structures as they provide where to begin reading information and how to interpret it. Using the information in the header we can then build the FAT (File Allocation Table) of the OLE document.

Offset	Data Type	Name	Comments
0	Char	magic[8]	Must equal 0x **d0 cf 11 e0 a1 b1 1a e1**
8	Char	clsid[16]	class id field is generally not used
24	Ushort	uMinorVersion	Minor version of the format: 33 is written by reference implementation. Used mainly for error checking purposes in a disc carving context.
26	Ushort	uDllVersion	major version of the dll format: 3 is written by reference implementation
28	Ushort	uByteOrder	indicates Intel byte-ordering
30	Ushort	uSectorShift	size of sectors in power-of-two (typically 9, indicating 512-byte sectors)
32	Ushort	uMiniSectorShift	size of mini-sectors in power-of-two (typically 6, indicating 64-byte mini-sectors)
34	Ushort	Reserved	reserved, must be zero
36	Ulong	reserved1	reserved, must be zero
40	Ulong	reserved2	reserved, must be zero
44	Ulong	num_FAT_blocks	number of SECTs in the FAT chain
48	Ulong	root_start_block	first SECT in the FAT Directory chain
52	Ulong	dfsignature	signature used for transactioning must be zero. The reference implementation does not support transactioning
56	Ulong	miniSectorCutoff	Maximum size for mini-streams: typically 4096 bytes.
60	Ulong	dir_flag	first SECT in the mini-FAT chain
64	Ulong	csectMiniFat	number of SECTs in the mini-FAT chain
68	Ulong	FAT_next_block	first SECT in the DIF chain
72	Ulong	num_extra_FAT_blocks	number of SECTs in the DIF chain
76	Ulong	sectFat[109]	FAT block list starts here. first 109 entries

Table 3. OLE Header Structure (After: Ref. [8])

Offset	Hexadecimal
0	**d0 cf 11 e0 a1 b1 1a e1** 00 00 00 00 00 00 00 00
16	00 00 00 00 00 00 00 00 3e 00 03 00 fe ff **09 00**
32	06 00 00 00 00 00 00 00 00 00 00 00 **01 00 00 00**
48	**5a 00 00 00** 00 00 00 00 00 10 00 00 5c 00 00 00
64	01 00 00 00 fe ff ff ff 00 00 00 00 59 00 00 00

Table 4. OLE Header Hexdump

The FAT contains the allocation information within a compound file. Every sector in the file is represented within the FAT in some fashion, including those sectors that are unallocated (free). The Fat is a virtual stream made up of one or more FAT Sectors [Ref. 8]. FAT sectors are arrays of SECT's that represent the allocation of space within the file. Each stream is represented in the FAT by a chain, in much the same fashion as a DOS file allocation table (FAT). To elaborate, the set of FAT sectors can be considered together to be a linked list—where each node in the list contains the SECT of the next sector in the chain, and this SECT can be used as an index into the Fat array to continue along the chain [Ref. 5].

Once the File Allocation Table is parsed, it is used to extract objects embedded within the file. This is done by examining the directory lists and then reading each entry within them. The entries themselves hold the application specific information we are looking for to determine what type of file it is (doc, ppt, xls…). The FAT is essentially an array of pointers to the directory listings which in turn are arrays of pointers to the entries themselves. The complexity of this hierarchy of pointers is the reason the Chicago Project developed the OLE API. Programmers need not learn the OLE file structure in order to achieve simple tasks of reading and writing to objects within the document. The entries can then be parsed and their name, size, and offset are stored to help determine the type of the file and size. Notice the listing in Figure 1 below which shows the output of a program called ole-dump which was written for the Chicago Project. It basically reads each entry of each directory structure and dumps the information to the screen. The OLE extraction algorithm uses the basic functions of this program to help discern the size and type of the file. Notice that DIRENT_2 has the title "WordDocument", all word documents contain some variation of this name as an object in one of their entries. Therefore it can be used as an identifier for Microsoft Word Documents.

```
DIRENT_0 :      root directory  Root Entry
prev dirent = ffffffff  next dirent = ffffffff  dir  block  = 3
unk1  = 20906   unk2  = 0       unk3  = c0
unk4  = 46000000        unk5  = 0       unk6  = 0
secs1  = 0      secs2  = 1896317920
days1  = 0      days2  = 29484230
start block  = 26
size  = 80
DIRENT_1 :      file    1Table
prev dirent = ffffffff  next dirent = 5 dir  block  = ffffffff
unk1  = 0       unk2  = 0       unk3  = 0
unk4  = 0       unk5  = 0       unk6  = 0
start block  = a
size  = 1000
DIRENT_2 :      file    WordDocument
prev dirent = 1 next dirent = ffffffff  dir  block  = ffffffff
unk1  = 0       unk2  = 0       unk3  = 0
unk4  = 0       unk5  = 0       unk6  = 0
start block  = 0
size  = 1222
DIRENT_3 :      file    0005    SummaryInformation
prev dirent = 2 next dirent = 4 dir  block  = ffffffff
unk1  = 0       unk2  = 0       unk3  = 0
unk4  = 0       unk5  = 0       unk6  = 0
start block  = 12
size  = 1000
DIRENT_4 :      file    0005    DocumentSummaryInformation
prev dirent = ffffffff  next dirent = ffffffff  dir  block  = ffffffff
unk1  = 0       unk2  = 0       unk3  = 0
unk4  = 0       unk5  = 0       unk6  = 0
start block  = 1a
size  = 1000
DIRENT_5 :      file    0001    CompObj
prev dirent = ffffffff  next dirent = ffffffff  dir  block  = ffffffff
unk1  = 0       unk2  = 0       unk3  = 0
unk4  = 0       unk5  = 0       unk6  = 0
start block  = 0
size  = 6a
Root Entry
1Table
4096
WordDocument                                          4642
SummaryInformation                                    4096
DocumentSummaryInformation                            4096
CompObj
106
```

Figure 1. ole-dump output of a MS Word Document

Figure 2 below shows the output of an Excel spreadsheet that has been run through the ole-dump program. DIRENT_1 is the main identifier here and it can be used to identify files generated by the Microsoft Excel program. Parsing the OLE File

Allocation Table provides a great advantage in being able to discern exactly what the contents of the file are.

```
DIRENT_0 :      root directory  Root Entry
prev dirent = ffffffff  next dirent = ffffffff  dir  block  = 2
unk1  = 20820   unk2  = 0       unk3  = c0
unk4  = 46000000        unk5  = 0       unk6  = 0
secs1  = 0      secs2  = 0
days1  = 0      days2  = 0
start block  = fffffffe
size  = 0
DIRENT_1 :      file    Workbook
prev dirent = ffffffff  next dirent = ffffffff  dir  block  = ffffffff
unk1  = 0       unk2  = 0       unk3  = 0
unk4  = 0       unk5  = 0       unk6  = 0
start block  = 0
size  = 33a6
DIRENT_2 :      file    0005    SummaryInformation
prev dirent = 1 next dirent = 3 dir  block  = ffffffff
unk1  = 0       unk2  = 0       unk3  = 0
unk4  = 0       unk5  = 0       unk6  = 0
start block  = 1a
size  = 1000
DIRENT_3 :      file    0005    DocumentSummaryInformation
prev dirent = ffffffff  next dirent = ffffffff  dir  block  = ffffffff
unk1  = 0       unk2  = 0       unk3  = 0
unk4  = 0       unk5  = 0       unk6  = 0
start block  = 22
size  = 1000
Root Entry
Workbook                                                    13222
SummaryInformation                                          4096
DocumentSummaryInformation                                  4096
```

Figure 2. ole-dump output of an Excel Spreadsheet

Lastly, Figure 3 shows an example of the contents of a simple Power Point Document with the unique identifier "Power Point Document" located in DIRENT_3. Notice that the size of each DIRENT is used to determine the actual size of the file, however, each size is contained within a block size that is specified in the OLE header, thus each entry must be padded to adhere to this structure.

```
DIRENT_0 :      root directory  Root Entry
prev dirent = fffffff  next dirent = fffffff  dir  block  = 2
unk1  = 64818d10       unk2  = 11cf4f9b       unk3  = aa00ea86
unk4  = e829b900       unk5  = 0       unk6  = 0
secs1  = 0      secs2  = 3860999472
days1  = 0      days2  = 29256468
start block  = 6
size  = 19c0
DIRENT_1 :      file    Current User
prev dirent = fffffff  next dirent = fffffff  dir  block  = fffffff
unk1  = 0       unk2  = 0       unk3  = 0
unk4  = 0       unk5  = 0       unk6  = 0
start block  = 66
size  = 38
DIRENT_2 :      file    0005    SummaryInformation
prev dirent = 1 next dirent = 3 dir  block  = fffffff
unk1  = 0       unk2  = 0       unk3  = 0
unk4  = 0       unk5  = 0       unk6  = 0
start block  = 36
size  = bcc
DIRENT_3 :      file    PowerPoint Document
prev dirent = fffffff  next dirent = 4 dir  block  = fffffff
unk1  = 0       unk2  = 0       unk3  = 0
unk4  = 0       unk5  = 0       unk6  = 0
start block  = 9
size  = b12
DIRENT_4 :      file    0005    DocumentSummaryInformation
prev dirent = fffffff  next dirent = fffffff  dir  block  = fffffff
unk1  = 0       unk2  = 0       unk3  = 0
unk4  = 0       unk5  = 0       unk6  = 0
start block  = 0
size  = 204
Root Entry
Current User
56
SummaryInformation                                     3020
PowerPoint Document                                    2834
DocumentSummaryInformation                             516
```

Figure 3. ole-dump output of an Power Point Document

Each of these documents has a very similar structure. They usually contain summary information which includes information about the author, the file name, when the file was last modified. Other methods to try to use the document summary information as a type of makeshift footer are not reliable as this information can appear at any location in the file.

The flexibility of the OLE file-structure also introduces the need for added error detection. OLE files are complex in nature and must be verified to ensure proper parsing and extraction. The consistency of various fields such fields as the block size of the document, the number of FAT blocks, and the mini-FAT cutoff can be used to perform error checking. This provides added assurance that the algorithm is not wasting its time parsing corrupted data.

The extraction of OLE files offers great promise. Because the Microsoft Office suite is so popular, documentation used by criminals can often be found in this format. This also enhances the forensic capabilities of the UNIX/LINUX platform as reliable OLE detection/extraction is only currently available on the Windows platform. In addition, with the advent of OpenOffice [Ref. 10] which provides support for the Microsoft Office suite these documents are often authored on UNIX systems as well. Thus this detection capability provides an invaluable resource to those performing forensic analysis.

2. PDF (Adobe Portable Document Format)

PDF is a file format used to represent a document in a manner independent of the application software, hardware, and operating system used to create it [Ref. 11]. A PDF file contains a PDF document and other supporting data. It is basically a binary file which also uses ASCII tags as delimiters to describe the header and trailer data structures in an SGML inspired fashion.

One of the main issues that earlier versions of Foremost had was that some formats (including PDF) often have multiple footers. This creates an obvious problem: how to determine which footer actually represents the end of the file. As a result Kornblum and Kendall developed a REVERSE search mechanism [Ref. 12] to allow them to find the last footer found in a given buffer. The REVERSE method essentially looked for the last footer in the buffer and associated it with the given header. This proved to be successful some of the time, but severely degraded its usefulness as the buffer size grew. Often multiple PDF files would be extracted as one file. In other cases, the footer appended was that of a corrupted PDF, causing the extracted file to be un-readable.

Further research of the PDF file specification revealed that a PDF contains multiple footers only if it has been "linearized". [Ref. 11] A linearized PDF file is one that has been organized in a special way to enable efficient incremental access in a network environment. Thus linearized PDF files are very common.

The PDF extraction function searches for the keyword “Linearized” in the header. If it is found, then the length of the file is stored in the header preceded by a “\L ” character sequence. This approach obviously increases the speed of Foremost as the program no longer needs to crunch through the entire PDF attempting to guess where it terminates. In this case, the function simply performs a search for the “\L” sequence and parses the number that follows, which is the file size in bytes. See Figure 4 for a structural description of a Linearized PDF.

.

```
Part 1: Header
                         %PDF-1.1
                         %  binary stuff

Part 2: Linearization parameters
                         43 0 obj
                         <<
                         /Linearized 1                                  version
                         /L 54567                                   file length
                         /H [475 598] Primary Hint Stream offset and length (Part 5)
                         /O 45       object number of first page's Page object (Part 6)
                         /E 5437                          offset of end of first page
                         /N 11                            number of pages in document
                         /T 52786     offset of first entry in main xref table (Part 11)
                         >>
                         endobj

Part 3: First Page xref table and trailer
                         xref
                         43 14
                         0000000052 00000 n
                         0000000392 00000 n
                         0000001073 00000 n
                         ...cross-reference entries for remaining objects in the first page...
                         0000000475 00000 n
```

Figure 4. Linearized PDF (From Ref. [11])

The PDF file format is more reminiscent of an XML document than a traditional binary document. This is why the common approach of being able to jump among data structures does not apply to this format. However, since linearized PDF files are becoming more prevalent, this algorithm will perform very quickly since the file size for this kind of file is often found within the first 100 bytes and no more file processing is necessary to extract these files which are often on the order of several megabytes in size.

Even when a file is not linearized the heuristic performs well in terms of successful extraction because of the unique trailer defined by the PDF specification (%%EOF). Hence a straight forward Boyer Moore search (described further in Chapter III) for the end of the file can be performed. This approach was successfully used to extract PDF's prior to PDF version 1.2 because the Linearized capability was not implemented.

Some minor error checking is also implemented. The first 100 bytes must include an "obj" tag, the fundamental storage tag for all PDF elements. An example of a non-linearized header is given below in Figure 5. Notice that the obj reference is still intact in this case making it a valuable marker to determine whether or not the file has been corrupted.

Offset	**Hexadecimal**	**ASCII**
00	25 50 44 46 2D 31 2E 33 0A 25 C7 EC 8F A2 0A 36	%PDF-1.3.%Çì□¢.6
16	20 **30 20 6F** 62 6A 0A 3C 3C 2F 4C 65 6E 67 74 68	0 **obj**.<</Length
32	20 37 20 30 20 52 2F 46 69 6C 74 65 72 20 2F 46	7 0 R/Filter /F
48	6C 61 74 65 44 65 63 6F 64 65 3E 3E 0A 73 74 72	lateDecode>>.str
64	65 61 6D 0A 78 9C AD 5A 49 73	eam.xœ-ZIs

Figure 5. Non Linearized Header

3. JPEG

JPEG stands for Joint Photographic Experts Group, which is a standardization committee. It also stands for the compression algorithm that was invented by this committee. To complicate things a bit more, JPEG compressed images are often stored in a file format called JFIF (JPEG File Interchange Format). JPEG data structures are composed of segments that are marked by identifiers [Ref. 13]. A listing of these markers is provided in Table 5. Each of these markers is preceded by a byte which equals "0xff". For example a common JPEG header may look like "0xff d8 ff e0 00 10 4a 46 49 46" (Hexadecimal), this is the simple case. The old method, implemented in earlier versions of Foremost, of grabbing a file based on header and footer information works well.

Marker Name	Marker Identifier	Description
SOI	0xd8	Start of Image
APP0	0xe0	JFIF application segment
APPn	0xe1 – 0xef	Other APP segments
DQT	0xdb	Quantization Table
SOF0	0xc0	Start of Frame
DHT	0xc4	Huffman Table
SOS	0xda	Start of Scan
EOI	0xd9	End of Image

Table 5. JPEG Marker Information (After: Ref.[13])

However, with the advent of digital cameras and the introduction of changes to the JPEG [Ref. 14] specifications, this method is no longer satisfactory. The new formats now allow for multiple headers, footers and even nested images, to support thumbnails for example. Digital cameras often utilize the APP segment marker "0xe1" to signify that they include more meta-data than the standard JFIF. Table 6 shows the hexadecimal representation of a JPEG taken from a Cannon digital camera; notice that the JPEG header repeats itself in the first block. The footers are also repeated for a total of 3 header/footer pairs in this specific file. Most tools that use the header/footer method of extraction, will extract three files out of this one image, one of those being a valid thumbnail while the others will appear as corrupt. For these reasons a more intelligent algorithm must be provided.

However, these compound formats still adhere to the common JFIF header structure. Thus even multiple headers and footers pose no problems to the implementation described below. Complex files can even increase the speed of the algorithm because, as more of the data can be skipped, less to be processed via the Boyer-Moore algorithm.

Offset	Hexadecimal View of JPEG Data
0	**ff d8 ff e0 00 10 4a 46 49 46** 00 01 02 01 00 48
10	00 48 00 00 **ff e1** 0b d5 45 78 69 66 00 00 4d 4d
20	00 2a 00 00 00 08 00 0a 01 0f 00 02 00 00 00 06
180	00 00 00 01 00 00 00 48 00 00 00 01 **ff d8 ff e0**
190	**00 10 4a 46 49 46** 00 01 02 01 00 48 00 48 00 00
be0	49 15 32 49 45 24 94 ff 00 **ff d9** ff ed 10 4c 50
1160	5f 00 18 00 01 **ff d8 ff e0 00 10 4a 46 49 46** 00
1bc0	ff 00 **ff d9** 00 38 42 49 4d 04 21 00 00 00 00 00

Table 6. Canon Digital Camera JPEG representation

The JPEG extraction algorithm exploits the fact that each JPEG marker contains the size of the header that the marker identifies. This allows the algorithm to jump from header to header until an invalid header is reached. If the file is a valid JPEG then the last marker parsed will be the SOS (Start of Scan) marker which signifies the beginning of the actual image data. Once this marker is reached then a Boyer Moore search for the "0xff d9" marker (which signifies the EOF) ensues.

With this ability to parse the JPEG data structures, our enhanced version of Foremost can now perform some error checking to ensure the file being extracting has not been corrupted. For instance each JPEG image must contain a Huffman Table marker as well as a Quantization Table, these checks are simple, efficient, and reduce the amount of information that the forensic examiner must process manually.

This method of extraction increases the accuracy of extraction as well as the speed as entire headers are skipped instead of being processed by the searching algorithm. Headers are kilobytes in size, so the fact that they are parsed rather than searched and interpreted byte by byte offers significant computational savings.

4. GIF

The Graphics Interchange Format (GIF) defines a protocol intended for the on-line transmission and interchange of raster graphic data in a way that is independent of the hardware used in their creation or display. There are two common versions of this format the 87a and 89a revision [Ref. 15]. This format has remained unchanged for the last decade and thus has proven to be a rather easy file to extract. It is one of the few which has a defined header and footer. Both of which occur only once in the file. Thus header and footer information is sufficient to successfully extract these files.

Table 7 illustrates header and footer information from a common GIF image. The GIF extraction algorithm searches for the unique string "\x47 \x49 \x46 \x38" (GIF8), once this is reached further tests are performed to determine if it is in fact a valid GIF file and whether it is revision 87a or 89a. Once this validation is performed a Boyer Moore search is ensues to find the unique "\x00 \x3b" identifier to determine the end of the GIF stream.

Offset	**Hexadecimal**
0	**47 49 46 38** 39 61 6c 02 22 03 a2 00 00 ff ff ff
	…
48e0	60 05 5c 02 00 **00 3b** 00

Table 7. GIF File Format

The only improvement we made to this extraction method is the fact that each version is analyzed in one pass through the data. Previous versions of Foremost would have to do independent searches for each header (87a and 89a). These are combined in the enhancement so search time is reduced by not analyzing the same information multiple times.

5. BMP (Windows Bitmap Files)

A BMP (Windows Bitmap File) [Ref. 16] is comparatively one of the more trivial files to successfully extract. Table 8 shown below illustrates the information provided in a BMP header. Notice the bfSize field in bold print, as this is the size of entire file in bytes. This is located at the offset 2 in the file! It may seem that extraction can be performed once this information is determined but additional checks must be provided to help ensure that the file being extracted is indeed valid BMP. The fact that header is only marked by two bytes "\x42 \x4d" (BM) means that a lot of false positives will be handed to the extraction function so a lot of "sanity" checking must be performed. Thus the horizontal and vertical sizes of the BMP are checked to see if they are reasonable values. If they are, then we have an added level of assurance that the file is indeed a Bitmap. More error checking could be added to take advantage of the data in the rather large header BMP files provide.

Offset	Field	Size	Contents
0000h	Identifier	2 bytes	'BM' - Windows 3.1x, 95, NT, …
0002h	File Size	1 dword	Complete file size in bytes.
0006h	Reserved	1 dword	Reserved for later use.
000Ah	BitmapData Offset	1 dword	Offset from beginning of file to the beginning of the bitmap data.
000Eh	Bitmap Header Size	1 dword	Length of the Bitmap Info Header used to describe the bitmap colors, compression, … The following sizes are possible: 28h - Windows 3.1x, 95, NT, … 0Ch - OS/2 1.x F0h – OS/2 2.x
0012h	Width	1 dword	Horizontal width of bitmap in pixels.
0016h	Height	1 dword	Vertical height of bitmap in pixels.
001Ah	Planes	1 word	Number of planes in this bitmap.
001Ch	Bits Per Pixel	1 word	Bits per pixel used to store palette entry information. This also identifies in an indirect way the number of possible colors. Possible values are:
001Eh	Compression	1 dword	Compression specifications. The following values are possible: 0 - none (Also identified by BI_RGB) 1 - RLE 8-bit / pixel (Also identified by BI_RLE4) 2 - RLE 4-bit / pixel (Also identified by BI_RLE8) 3 - Bitfields (Also identified by BI_BITFIELDS)
0022h	Bitmap Data Size	1 dword	Size of the bitmap data in bytes. This number must be rounded to the next 4 byte boundary.
0026h	HResolution	1 dword	Horizontal resolution expressed in pixel per meter.
002Ah	VResolution	1 dword	Vertical resolution expressed in pixels per meter.
002Eh	Colors	1 dword	Number of colors used by this bitmap. For a 8-bit / pixel bitmap this will be 100h or 256.
0032h	Important Colors	1 dword	Number of important colors. This number will be equal to the number of colors when every color is important.
0036h	Palette	N * 4 byte	The palette specification. For every entry in the palette four bytes are used to describe the RGB values of the color in the following way:
0436h	Bitmap Data	x bytes	Depending on the compression specifications, this field contains all the bitmap data bytes which represent indices in the color palette.

Table 8. BMP Header Information(After: Ref. [16])

An example of a bitmap header is given in Table 9 showing that the file size according the bytes 2-6 is 163,878 which has the hexadecimal representation "0x26 0x80 0x02 0x00" in little endian format. Also highlighted are the horizontal and vertical sizes of the BMP located at offsets 18 and 22 in the file. With this information we can deduce that the Bitmap is 400x407 pixels which is a reasonable value for a bitmap image. As noted previously these are invaluable for error detection.

Offset	Hexadecimal
0	**42 4d** ***26 80 02 00*** 00 00 00 00 36 04 00 00 28 00
16	00 00 **90 01 00 00 97 01 00 00** 01 00 08 00 00 00
32	00 00 f0 7b 02 00 20 2e 00 00 20 2e 00 00 00 01
48	00 00 80 00 00 00 00 00 00 00 73 73 73 00 29 23
64	28 00 ce be bf 00 b5 a2 a5 00 0f 09 0e 00 52 4c
80	51 00 9d 8a 8d 00 49 39 3a 00 d7 d4 d0 00 62 5c

Table 9. BMP Header in hexadecimal

The previous version of Foremost would merely check for the BM header and then dump the next 50KB into a file and make the examiner determine the EOF. The current implementation is an obvious improvement as the examiner can simply look at the files content in an image viewing application as opposed to trying to interpret hexadecimal values and deduce file specific information from them. This drastically reduces the examiners workload because the majority of data they are looking for may be graphical in nature, especially in cases where pornography is involved.

6. MOV (QuickTime Movie files)

A QuickTime file [Ref. 17] is simply a collection of atoms, the basic data structures of the file. QuickTime does not impose any rules about the order of these atoms. This allows for ease of concatenation when editing movie files. See Figure 6 for a typical structure of a QuickTime movie file.

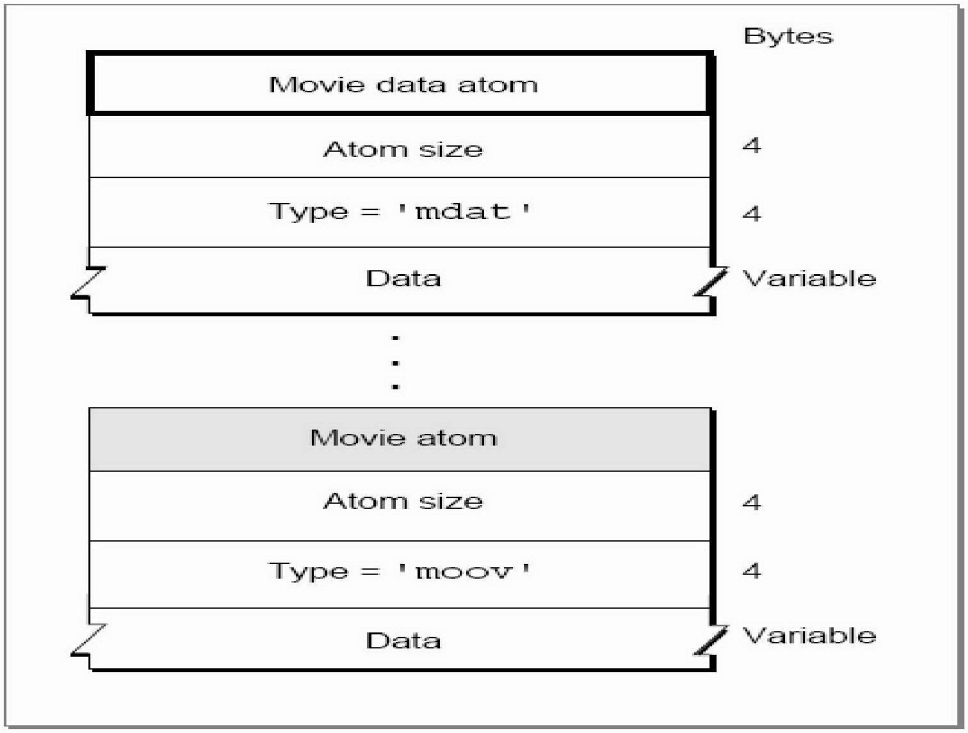

Figure 6. QuickTime Movie Structure (From: Ref. [17])

This modular file format provides flexibility to the application but is difficult to parse using traditional methods, as no unique marker is used to signify where the file terminates. However, if the atoms are parsed, the size of each atom is included in the atom header. This provides the ability to jump from atom header to atom header until an invalid header is reached. Once this occurs, the EOF has been determined. This method also is highly optimized as MOV files are often large.

Another problem, the flexibility QuickTime files creates, is the fact that the structure of the headers can vary somewhat. The standard atom header is of type 'moov' but modern digital cameras implement what is called VJPEG (Video JPEG) format which uses the atom type 'pnot' as the first atom in the file. For this reason these two extraction methods are performed separately but will both be invoked when searches for "multimedia" files are performed.

Notice from Figure 6 that the same format of length, then type, and then data (value) is used as the basic structure of an atom. Table 10 provides a step by step walk through of how a MOV file is parsed through iteration of the extraction function. The test case was a VJPEG file that was 9,275,716 bytes in size. Each iteration shows that the first four bytes of the header contains the header size in big endian format while the remaining four bytes contain the type of the atom in ASCII text.

The first iteration determines main header information located at offset 0. The size of the header is extracted (in this case 20) and then the file pointer is moved accordingly to offset 20. At offset 20 a PICT atom is located and is 6196 bytes in size. Jumping again to offset 6216 is the main data portion of the MOV file which is of type “mdat”. All valid MOV files must contain this atom; therefore, it is used as an error checking mechanism to determine if the file to be extracted is intact. The last atom is of type “moov” which is the standard header for most MOV files, but as shown here can be included anywhere in the file. Jumping the size of the “moov” atom puts the file pointer at the end of the file. Summing the size of each atom yields an original file size of 9,275,716 bytes.

Header#	**Size**	**Type**	**Header in Hexadecimal**
0	20	pnot	0 0 0 14 70 6e 6f 74
1	6196	PICT	0 0 18 34 50 49 43 54
2	9266184	mdat	0 8d 64 8 6d 64 61 74
3	3316	moov	0 0 c f4 6d 6f 6f 76
Total	9275716		

Table 10. MOV Extraction Algorithm Step-through

7. WMV (Windows Media Video)

Windows Media Video/Windows Media Audio files use the ASF file format[Ref. 18]. The Advanced Streaming Format (ASF) is an extensible file format designed to store synchronized multimedia data. It supports data delivery over a wide variety of networks and protocols while still proving suitable for local playback. ASF supports advanced multimedia capabilities including extensible media types, component download, scaleable media types, author-specified stream prioritization, multiple language support, and extensive bibliographic capabilities, including document and content management.

The invaluable (for us) data structure in the ASF format contains the header "0xA1 DC AB 8C 47 A9". The structure beginning with this header contains the file properties object header and the file size (in bytes). Thus it can be used to determine the EOF. This header is often found within the first 512 bytes of the file and thus processing often extremely large WMV/WMA files is avoided. See the Figure 7 below for a description of the ASF format.

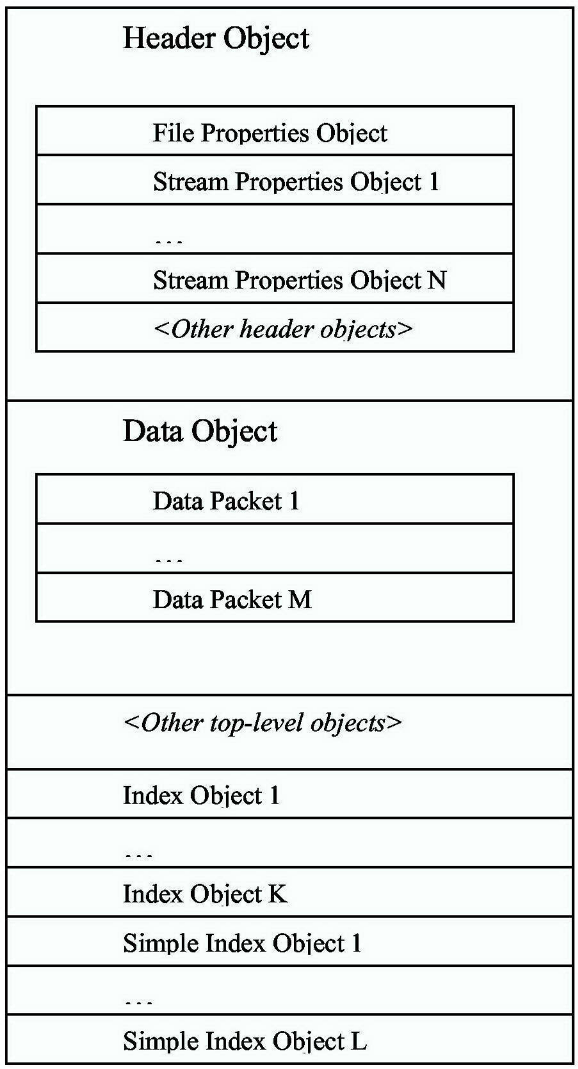

Figure 7. ASF File Structure (From: Ref. [18])

The basic idea behind the algorithm we use to parse these files is that once the file header is found ("\x30\x26\xB2\x75\x8E\x66\xCF\x11"), a search for the file properties header which contains the file size information is executed. This is shown in Table 11 below. Once the file properties object has been located, the file size is located at offset 40 within the object. This information helps determine the end of the file. The trick is that the file properties object can be located at various offsets throughout the beginning of the file, which is why a search for the header ID must be used to determine its location.

Name	**Size (bytes)**
Object ID	16
Object Size	8
File ID	16
File Size	**8**
Creation Date	8
Data Packets Count	8
Play Duration	8
Send Duration	8
Preroll	8
Flags	4
Minimum Data Packet Size	4
Maximum Data Packet Size	4
Maximum Bitrate	4

Table 11. ASF File Properties Object Structure (After: Ref. [18])

See Table 12 below for a sample header of a WMV file and the import values in bold. Once the file properties object is found at offset 69, we know from the structure of the file properties object that the file size is stored at offset 109 in an eight byte value in little endian format. Thus a simple pointer addition can be used to arrive at the file size of the WMV.

Offset	Hexadecimal
0	**30 26 b2 75 8e 66 cf 11** a6 d9 00 aa 00 62 ce 6c
16	d3 03 00 00 00 00 00 00 09 00 00 00 01 02 ce 75
32	f8 7b 8d 46 d1 11 8d 82 00 60 97 c9 a2 b2 26 00
48	00 00 00 00 00 00 02 00 01 00 90 47 00 00 02 00
64	18 4b 01 00 **a1 dc ab 8c 47 a9** cf 11 8e e4 00 c0
80	0c 20 53 65 68 00 00 00 00 00 00 00 d7 51 ed 1c
96	16 91 5e 4a bd db fe 9e eb 31 e3 da **98 1b 6c 00**
112	00 00 00 00 e0 03 79 3e d0 c6 c1 01 75 15 00 00

Table 12. ASF Header in Hexadecimal

This added knowledge of the internal data structures has provided us the means to enhance Foremost so that it can avoid searching through, in many cases, megabytes of information to determine a files endpoint. Previous versions of Foremost provide no support for WMV/WMA files. This new ASF capability opens the door to the multimedia files. This will aide in the prosecution of pornography cases. In addition, with the increasing popularity of WMA files for pirating music Foremost 1.0 can be a useful tool in the prosecution of copyright violations.

8. ZIP

Zip files often contain multiple embedded files of varying formats; these are structured in an incremental fashion, followed by a “central directory structure”. ZIP archives are a standard format for compressing and storing multiple files. Each file contained within the zip has its own valid ZIP header with its compressed and uncompressed data sizes stored within it. This information can be exploited to increase the speed of the extraction of ZIP files.

```
[local file header 1]
[file data 1]
[data descriptor 1]
.
.
.
[local file header n]
[file data n]
[data descriptor n]
[central directory]
[zip64 end of central directory record]
[zip64 end of central directory locator]
[end of central directory record]
```

Figure 8. Basic Zip File Structure (From Ref. [19])

The algorithm works incrementally by parsing each local file header, using the compressed size field located at offset 20. This value is then used to jump to the next local file header. Once all the files headers have been analyzed then a Boyer Moore search for the identifier of the end of the central directory record is conducted. Once this object is located (the structure is given in Table 13) the algorithm then reads the length of the comment field and jumps to that value plus the size of the object (20 bytes). The result is the end of the zip file.

Field Description	**Size**
central file header signature	4 bytes (**0x02014b50**)
version made by	2 bytes
version needed to extract	2 bytes
general purpose bit flag	2 bytes
compression method	2 bytes
last mod file time	2 bytes
last mod file date	2 bytes
crc-32	4 bytes
compressed size	**4 bytes**
uncompressed size	4 bytes
filename length	2 bytes
extra field length	2 bytes
file comment length	2 bytes
disk number start	2 bytes
internal file attributes	2 bytes
external file attributes	4 bytes
relative offset of local header	4 bytes

Table 13. ZIP local file header structure (From Ref.[19])

The structure an outline of the central directory structure, is given below in Table 14. The header value "0x50 4b 05 06" is used to flag the beginning of the structure at which point the comment field is extracted to determine the exact EOF.

Field Description	**Size**
end of central dir signature	4 bytes **(0x06054b50)**
number of this disk	2 bytes
number of the disk with the start of the central directory	2 bytes
total number of entries in the central directory on this disk	2 bytes
total number of entries in the central directory	2 bytes
size of the central directory	4 bytes
offset of start of central directory with respect to the starting disk number	4 bytes
.ZIP file comment length	**2 bytes**
.ZIP file comment	(variable size)

Table 14. End of Central Directory Object Structure (From Ref.[19])

A sample run through of the algorithm is provided below with a zip archive containing 9 files with a total size 679168 bytes. As show in the Table 15 each iteration jumps to the next file in the archive. A total of 10 iterations are required because of the initial zip file header. Each file size is the summation of the compressed file size (located at offset 20 within the local file header as show in Table 13), the file name length, the extra length, and the size of the data structure itself (30 bytes). These ten jumps amount to a total file size of 678439 bytes, there are some peripheral data structures at the end of the file so a Boyer Moore search is done to find the end of the central directory structure. This proves trivial as 729 bytes remain after the jump loop takes place, thus the vast majority of the search overhead is avoided.

Header#	Size	Header in Hexadecimal
0	65002	**50 4b 3 4 14** 0 0 0 8 0 34 87 30 32 12 16
1	27041	**50 4b 3 4 14** 0 0 0 8 0 34 87 30 32 49 e2
2	20516	**50 4b 3 4 14** 0 0 0 8 0 34 87 30 32 ed 65
3	186436	**50 4b 3 4 14** 0 0 0 8 0 34 87 30 32 c5 15
4	17202	**50 4b 3 4 14** 0 0 0 8 0 34 87 30 32 2d 72
5	259494	**50 4b 3 4 14** 0 0 0 8 0 34 87 30 32 06 0f
6	39482	**50 4b 3 4 14** 0 0 0 8 0 34 87 30 32 33 b1
7	55707	**50 4b 3 4 14** 0 0 0 8 0 35 87 30 32 25 6f
8	7143	**50 4b 3 4 14** 0 0 0 8 0 35 87 30 32 47 77
9	416	**50 4b 3 4 14** 0 0 0 8 0 35 87 30 32 21 8b
Total	678439	

Table 15. ZIP extraction algorithm step-through

Obvious improvement can be seen implementing this extraction method as opposed to traditional methods. Much of the searching burden is relieved by the ability to merely jump to each file objects until only a few small data structures remain to parse. Comparing this to the previous method which Foremost used is not even worthy of comparison as the speed of extraction is increased exponentially. Since zip files have no well defined footer, the examiner was previously forced to attempt to determine where the file ended by incrementally extracting more of the file and running zip decompression algorithms against it. This is time consuming and should be avoided if possible.

9. GZIP

The GZIP file format is recursive in nature as it is merely processed until the decompression algorithm completes. No notion of the original data size is given to the algorithm, therefore in order to fully support the extraction of GZIP files a decompression algorithm must be incorporated into Foremost. Currently this introduces system dependent issues and is left as future work. However, a simplified extraction method is possible with marginally good results.

The GZIP header value is equal to "0x1f 8B" which is used to identify the file as a gzip file. This identifier is followed by a one byte value which identifies the compression method used in the file. CM = 0-7 are reserved. CM = 8 denotes the "deflate" compression method, which is the one customarily used by gzip. If this information is parsed and verified to be a gzip header with some degree of assurance we can then jump to the end of the header and search for the string "\x00 \x00 \x00 \x00". This works reasonably well as files in the GZIP format do not write blocks of zero's in the data portion of the file, however empty sectors on the disc usually contain all zeros. The fact that this often overshoots the end of the file in most cases is irrelevant since the decompression algorithm ignores extraneous information, the file will inflate without a problem.

Offset	Hexadecimal
0	**1F 8B** 08 08 E6 38 BA 3B 00 03 69 74 73 34 2D 31

Table 16. GZIP Header in Hexadecimal

The GZIP file format is most common on a UNIX platform and is therefore a valuable commodity to open source tools that analyze such environments. This algorithm while still a "best effort" provides support for a format that was not supported in older versions of Foremost. The method is still the same in terms of using basic header and footer information to deduce the file size. However, the error checking capabilities in terms of checking to see if the header contains reasonable values significantly reduces the number of false positives generated by the program.

10. RIFF

The RIFF file structure is used by various file formats, most notably AVI (Audio/Video Interleaved) and WAV. The WAV File Format is a file format for storing digital audio (waveform) data. This format is very popular as it is most commonly used in commercial music cd's. It is also widely used in professional programs that process digital audio waveforms. WAVE files are often just RIFF files with a single "WAVE" chunk which consists of two sub-chunks -- a "fmt" chunk specifying the data format and a "data" chunk containing the actual sample data [Ref. 20]. Table 17 below shows a sample WAV header stored in the RIFF file structure. The first four bytes indicate the RIFF file structure, followed by the file size stored in little endian, and finally the WAV signature indicating that this is indeed a WAV file within the RIFF structure.

Offset	Hexadecimal	Ascii
00	**52 49 46 46 B0 A3 01 00 57 41 56 45** 66 6D 74 20	**RIFF°£..WAVE**fmt
16	10 00 00 00 01 00 02 00 44 AC 00 00 10 B1 02 00	........D¬...±..
32	04 00 10 00	....

Table 17. Wave File Header

The Audio/Video Interleaved (AVI) file format is a RIFF file specification used with applications that capture, edit, and playback audio/video sequences. In general, AVI files contain multiple streams of different types of data. Most AVI sequences will use both audio and video streams[Ref. 21]. The AVI RIFF form is identified by the four-character code "AVI " as noted below in Table 18. All AVI files include two mandatory LIST chunks. These chunks define the format of the streams and stream data and are also used to provide an error checking mechanism to the extraction function.

Offset	Hexadecimal	Ascii
00	**52 49 46 46** 8A E7 86 09 **41 56 49** 20 4C 49 53 54	RIFFŠç†.**AVI** LIST
16	26 01 00 00 68 64 72 6C 61 76 69 68 38 00 00 00	&...hdrlavih8...
32	6B 04 01 00 C3 DA 34 00 00 00 00 00 10 08 00 00	k...ÃÚ4.........

Table 18. AVI File Header

Extracting an AVI/WAV file is trivial because the file size is stored in the RIFF file format at offset 4. Thus minimal error checking is required to ensure that the file is indeed an AVI or a WAV before it is extracted. This type of error checking includes, in the case of an AVI, the verification that the header contains the LIST chunk a mandatory portion the file specification.

This algorithm is an obvious improvement over the previous version of Foremost as WAV files were not supported and AVI files do not contain valid footers, therefore as we have seen previously the burden is then needlessly placed upon the examiner. Also the speed of the extraction function is a major enhancement as only the first block of data needs to be analyzed to determine the actual file size.

11. HTML

Extracting HTML (Hyper Text Markup Language) files requires the challenging tasks of building heuristics to look at file content as opposed to its data structures. HTML files are fairly intuitive to extract as they have a defined footer. This is not the case with most other ASCII files. The main problem in dealing with HTML is the generation of false positives. To deal with this the new extraction algorithm checks the first block of the file to ensure it is indeed ASCII printable, this greatly reduces the number of false positives without checking the entire file byte by byte. With the advent of XML and CGI scripts, it is not uncommon to see HTML headers within files that are not HTML at all. Some would argue that these CGI scripts and other binary files are valuable evidence, thus the traditional method of extraction based on strictly header/footer data is available via the configuration file of Foremost.

This method of error detection is somewhat slower than just looking at the header/footer pair. However, much time is saved by not having to sift through files which do not appear to be HTML, thus increasing the productivity of the examiner. Through experimentation it was found that often small portions of CGI scripts are extracted that only contain the "<html>" and "</html>" tags embedded within the binary values. The utility of such files is minimal and thus this algorithm attempts to rectify this problem.

12. CPP (C/C++ Source Code)

Source code detection could be a useful weapon in the prosecution of hackers because Foremost could potentially recover some source code that a hacker compiled on a "victim's machine". The detection of C source code is an intriguing task as no well defined header or footer exists for these types of ASCII text files. Thus a system of markers and keyword searches is the best method for building a system which can intelligently extract these files. The fundamental marker that the CPP extraction algorithm uses is the "#include" statement which must be in source code if it wishes to use any libraries whatsoever. However this isn't fool proof; as a C file may contain only function definitions and be included or linked with another file that contains the #include statement. In addition many local exploits are short and usually only consist of a single source file. This is a limitation that is accepted as this extraction method can be termed as a "best effort" method.

Once the first marker is found then the file is scanned to until a non ASCII printable character is reached. With this new buffer of information a series of keywords is then searched for to give added confidence that the data is in fact source code. Other keywords include "int ", "char ", and "#define ", these strings help build a "score" for the data and if the file meets the minimum score threshold then it is extracted and deemed source code. This method works reasonably well but a more sophisticated system must be implemented to "catch all" of these types of files.

B. SEARCH ALGORITHMS

1. Boyer Moore Description

The Boyer-Moore searching algorithm, described in R. S. Boyer and J. S. Moore's 1977 paper "*A Fast String Searching Algorithm*" [Ref. 22] is among the best ways known for finding a substring in a search space. Using their method it is possible to search a data space for a known pattern without having to examine all the characters in the search space. This is why it was chosen as the fundamental searching algorithm employed by Foremost. Boyer-Moore search algorithms are based on two search heuristics.

The first of these rules tell us how to search for substrings without repeats in a data space. Keep a pointer into the data space at the current search location; initialize this pointer to the start of the space plus n - 1 characters where n is the number of characters in the target string. Compare the character in the data space pointed to by this pointer with the characters in the target string. If this character does not occur in the target string, advance the pointer by n places. If the character does occur in the target string, advance the pointer by n - p places where p is the position that the character in question first occurs in the target string. This process repeats until either a match is found or we have shifted past the end of the search space.

The second search heuristic applies to searching for targets with repeating patterns. Using only the rules set forth in the first heuristic will work for targets with repeating patterns, but the search will not be as efficient as possible. By examining partial matches and repeats in the target string, though, it is possible to make more drastic pointer jumps and arrive at the match more rapidly. This type of jump is based on a table which is computed before the search begins.

Figures 9 & 10 show the improvements Boyer-Moore makes when compared with brute force search methods. Notice that Boyer-Moore completes the search in less than half the comparisons of the conventional methods.

```
For any commands
|                  compare (a to F)
and
 |                 compare (a to o)
 and
  |                compare (a to r)
  and
   |               compare (a to space)
   and
    |||            compare (a to a), (n to n) and (d to y)
    and
     |             compare (a to n)
     and
      |            compare (a to y)
      and
       |           compare (a to space)
       and
        |          compare (a to c)
        and
         |         compare (a to o)
         and
          |        compare (a to m)
          and
           |       compare (a to m)
           and
            |||    compare (a to a), (n to n) and (d to d)
            and
                   ---------------------------------------
                   Total of 17 comparisons.
```

Figure 9. Brute Force Search (From Ref. [23])

```
For any commands
   |               compare (d to r)
and
      |            compare (d to n)
   and
       |           compare (d to y)
    and
          |        compare (d to o)
       and
             |     compare (d to a)
          and
             |||   compare (d to d), (n to n) and (a to a)
             and
                   ---------------------------------------
                   Total of 8 comparisons.
```

Figure 10. Boyer Moore Search (From Ref. [23])

2. Algorithm Analysis

An analysis of Boyer-Moore shows that vast improvements can be achieved versus the brute force O(n^2) method. M is equal to the size of the search space and n is equal to the size of the string. The preprocessing phase has $O(m+\sigma)$ time and space complexity, the searching phase has O(mn) time complexity, 3*n text character comparisons in the worst case when searching for a non periodic pattern, and O(n / m) best case performance[Ref. 22]. This added performance is the reason this algorithm is the most popular for performing text searches in many editors, but it also suits the disc carving purpose because it can be adapted to perform hexadecimal searches as well.

C. INDIRECT BLOCKS

1. UNIX File System Overview

As with other operating systems, files are not necessarily written to disk contiguously by UNIX file-systems. A file may be stored in several different blocks, seemingly randomly chosen; however, the blocks do generally adhere to a semi-contiguous structure. UNIX creates a data structure called an inode to maintain all relevant information about a file, including which disc blocks the file has been stored on. Each inode is stored sequentially in an array, so the inode itself does not affect its corresponding file size. The file system is retrieved during the boot process. The boot process contains a hard coded inode number, which represents a file location containing a boot block in memory and inode list [Ref. 24].

UNIX deals with fragmentation by redirecting its inodes. It creates "indirect blocks" for those inodes pointing to large files, where the file is stored in non-contiguous blocks on a disk. Those indirect blocks contain the addresses of the blocks containing the file, and the inode in turn contains the address of that indirect block.

2. Indirect Block Detection

Indirect block detection is an invaluable tool in successful extraction of files from a UNIX/LINUX file system. Indirect blocks are used when a file consists of more than twelve blocks and the file system needs to store additional information so that it can keep track of all the blocks allocated to the file. The ability to detect indirect blocks, use the information stored in those blocks greatly increases the detection and extraction capabilities in UNIX file systems. Figure 11 depicts a screenshot from the debugfs program which shows the blocks that are allocated to a Power Point file. Notice that the file is larger than 12 blocks, thus it requires the usage of an indirect block (IND) located at offset 8525813. In this case, as is often the case, IND is contiguous with the rest of the blocks; however for extraction purposes it must be detected and removed.

```
debugfs:  stat intro.ppt
Inode: 4156452   Type: regular    Mode:  0755   Flags: 0x0   Generation: 3244518
08
User:     0   Group:     0   Size: 98816
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 208
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x41945dd4 -- Thu Nov 11 22:53:08 2004
atime: 0x41a0ef5f -- Sun Nov 21 11:41:19 2004
mtime: 0x408d49b0 -- Mon Apr 26 10:41:04 2004
BLOCKS:
(0-11):8525801-8525812, (IND):8525813, (12-24):8525814-8525826
TOTAL: 26

debugfs:
```

Figure 11. Debugfs Screenshot

Using the UNIX program dd, we can view the structure of the indirect block. Figure 12 shows the actual indirect block used in the example Power Point file. Notice that each 4 byte chunk is the location of the remaining blocks allocated to the file. The file system uses this information in order to rebuild the file before giving it to the operating system. The fact that these blocks are usually increasing and fairly close together can be exploited by a heuristic function which detects and removes indirect blocks. The algorithm works by first analyzing the structure of the indirect block to verify that it is not simply part of the file. Verification of increasing offsets followed by a variable amount of 0's occurs. Then the differences between each offset is checked to determine whether they exceed a given threshold value to add assurance that the block under study is an indirect block. If a difference exceeds one, meaning that the offsets are not contiguous then following logic ensues in an attempt to rebuild the file before it is handed off to the extraction algorithms.

```
00000000   F6 17 82 00  F7 17 82 00  F8 17 82 00  F9 17 82 00   ................
00000010   FA 17 82 00  FB 17 82 00  FC 17 82 00  FD 17 82 00   ................
00000020   FE 17 82 00  FF 17 82 00  00 18 82 00  01 18 82 00   ................
00000030   02 18 82 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000040   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000050   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000060   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000070   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000080   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000090   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000000A0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000000B0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000000C0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000000D0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000000E0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
000000F0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000100   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000110   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000120   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000130   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000140   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000150   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
00000160   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00   ................
---  ind.dd   --0x0/0x1000------------------------------------------------
```

Figure 12. Indirect Block Screenshot

One of the main problems with indirect block detection is the fact that often tools like Foremost are used on fragments of a disc. These may include just unallocated space, slack space, or maybe only a portion of a valid file-system is recoverable. In any case the offsets located in the indirect block cannot be trusted as they only hold true if the entire file-system is intact. Therefore, some assumptions must be made in order to attempt the reconstruction of non-contiguous files that contain indirect blocks. The first offset listed in the indirect block is assumed to be one more than that of the indirect block itself therefore all other offsets can be used relative to that one. Essentially the heuristic uses the remaining offsets as offsets from the first block listed in the indirect block. This works reasonably well as many indirect blocks that are not contiguous usually only contain one or two blocks that are not in order. Thus as long as the first block listed is contiguous, the algorithm performs with great success.

Another problem is the fact that the block size isn't the same across various UNIX file sizes. Thus the most common block size of 4096 bytes is tried first to see if the block meets the detection algorithm's specifications. Failing that, then other common block sizes must be tried in order to attempt to determine what the actual block size is. This may also be accomplished by having a user defined block size, if the user knows the specific file system used, such as UFS, EXT2, or EXT3.

File system vendors often trump security for speed. This is why files are often not overwritten when they are deleted but merely have their meta-data moved to unallocated space. The EXT3 file system actually does delete the inodes and indirect blocks of a file. Some may argue that this trend negates the need for indirect block detection. But the heuristic if often useful in the case where only a portion of the file-system may be recoverable, thus leaving some indirect blocks in tact.

No data carving tool has addressed the need for indirect block detection. On UNIX file-systems the advantages are huge as, files such as office documents, multimedia, archive, and even images routinely use more than 12 blocks, thus extraction algorithms will fail. As operating systems such as Linux increase in popularity the use of EXT2/EXT3 file-systems will increase so and so will the need for these types of algorithms which can interpret the data stored in indirect blocks. See the indirect block section in chapter four for examples of how indirect block detection improves successful extraction of various files.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. EXPERIMENTAL RESULTS

A. OVERVIEW

The tools used in this comparison include FTK, ILOOK, the original version of Foremost, and the modifications to Foremost presented in this paper. For testing purposes the implementation of the tool described in this paper will be referred to as Foremost 1.0.

Version 1.5 of FTK, the version current at the writing of this document, supports the following file formats: BMP, GIF, JPEG, EMF, PDF, HTML, AOL, and OLE. The capabilities of this product performed very well in experimental test cases. However, FTK only allows for carving of unallocated space thus it will not be used in the test cases as it wouldn't provide a fair comparison with products that analyze an entire disc image. However, the tool seems to use an approach similar to the one described in this paper.

ILOOK supports far more file formats than FTK but with varying success. It even provides multiple versions of extraction algorithms for the same file format. This is mainly because ILOOK can incorporate new file signatures into its data carving mechanism. For example, ILOOK contains three different extraction methods for carving JPEG files. Testing these algorithms showed that they perform relatively well but they do not catch everything and they perform at varying speeds. Overall this tool performs well but it definitely emphasizes quantity of output over quality. This can potentially become burdensome for an analyst.

B. NTFS

Brian Carrier, the main Sleuthkit developer, created a 10MB test image for testing forensic tools ability to extract jpeg data. The image is an NTFS partition containing the files listed in Table 19 below. Through experimentation it was discovered that NTFS does a very good job of storing files in contiguous memory blocks. This makes the disc carving process much easier than dealing with the indirect blocks of UNIX files systems. The MD5 of the image is "9bdb9c76b80e90d155806a1fc7846db5" and it can be downloaded at http://dftt.sourceforge.net/test8/. This image was used because of its availability and to demonstrate the utility of the JPEG algorithm described previously in addition to ZIP, GZIP, and OLE extraction capabilities.

Num	Name	MD5	Note
1	alloc\file1.jpg	75b8d00568815a36c3809b46fc84ba6d	A JPEG file with a JPEG extension
2	alloc\file2.dat	de5d83153339931371719f4e5c924eba	A JPEG file with a non-JPEG extension
3	invalid\file3.jpg	1ba4e91591f0541eda255ee26f7533bc	A random file with a JPEG extension
4	invalid\file4.jpg	c8de721102617158e8492121bdad3711	A random file with 0xffd8 as the first two bytes (the JPEG header signature). There is no JPEG footer or other header data.
5	invalid\file5.rtf	86f14fc525648c39d878829f288c0543	A random file with the 0xffd8 signature value in several locations inside of the file.
6	del1\file6.jpg - MFT Entry #32	afd55222024a4e22f7f5a3a665320763	A deleted JPEG file with a JPEG extension.
7	del2\file7.hmm - MFT Entry #31	0c452c5800fcfa7c66027ae89c4f068a	A deleted JPEG file with a non-JPEG extension.
8	archive\file8.zip	d41b56e0a9f84eb2825e73c24cedd963	A ZIP file with a ZIP extension and a JPEG picture named file8.jpg inside of it.
	file8.jpg	f9956284a89156ef6967b49eced9d1b1	A JPEG file that is inside of a ZIP file with a ZIP extension.
9	archive\file9.boo	73c3029066aee9416a5aeb98a5c55321	A ZIP file with a non-ZIP extension and a JPEG picture named file9.jpg inside of it.
	file9.jpg	c5a6917669c77d20f30ecb39d389eb7d	A JPEG file that is inside of a ZIP file with a non-ZIP extension.
10	archive\file10.tar.gz	d4f8cf643141f0c2911c539750e18ef2	A gzipped tar file that contains a JPEG picture named file10.jpg.
	file10.jpg	c476a66ccdc2796b4f6f8e27273dd788	A JPEG file that is inside of a gzipped tar file.
11	misc\file11.dat	f407ab92da959c7ab03292cfe596a99d	A file with 1572 bytes of random data and then a JPEG picture. This was created using the '+' option in the Windows copy.exe tool.
12	misc\file12.doc	61c0b55639e52d1ce82aba834ada2bab	A Word document with the JPEG picture inside of it.
13	misc\file13.dll:here	9b787e63e3b64562730c5aecaab1e1f8	A JPEG file in an Alternate Data Stream.

Table 19. Brian Carriers JPEG test image files (From Ref. [25])

After running ILOOK against the image the following files were extracted. Note that since the image is a valid NTFS partition ILOOK has the capability to mount the image and extract files via the meta-data. Note that this is not relevant to the disc carving capability of the tool. ILOOK uses a customizable database of file signatures to "carv" data. In essence it takes the same approach as Foremost 0.69 in that only header and footer data seems to be analyzed. Although this cannot be verified without the source code it seems that ILOOK uses a file size limit of 102,400 bytes for JPEG which explains why all files greater than that threshold were truncated.

Num	Name	MD5	Size	Note
1	530.jpg	f41b83ecabe49a70 752dca82020f2e3b	102,400	This file is truncated
2	1066.jpg	ad869aa50da6e2976562 b2fb9356b12b	102,400	This file has been truncated
3	1705.jpg	a5131a3a619edcdcd15d 2c134ad41da7	102,400	Truncated Picture #3
4	6688.jpg	b957180a0b411aba6b2 e9a9f0d68bdc6	102,400	This file has been truncated
5	10056.jpg	dac28876682e92996de 3b4aaa5bdf96b	26,112	Valid JPEG #2
6	10810.zip	a795b3d16f47a03f4f7f5 54b84ee3949	335,360	Corrupted
7	11466.zip	7d9e23b2e48f768f46a4 27de9d50e949	294,400	Valid archive containing picture #6

Table 20. ILOOK results from NTFS sample image

After running the traditional version of Foremost (0.69) the following files were extracted (See Table 21 below). The older version of Foremost performed reasonably well against an image consisting mostly of simple jpeg files. In addition since only one OLE document was included in the image Foremost 0.69 was able to extract the Word Document using its NEXT search capability. The NEXT search capability allows Foremost to use the header as the footer, this approach relies on the fact that OLE documents are often written in relatively close memory space. Since they are sometimes written in groups the header of the NEXT document can be used to determine the EOF of the current document. This method works pretty well for small images but severely degrades as images grow and documents become more spread out.

Num	Name	MD5	Size	Note
1	00000000.jpg	75b8d00568815a36c380 9b46fc84ba6d	274260	Valid picture #1 (Matches MD5)
2	00000001.jpg	0c452c5800fcfa7c66027 ae89c4f068a	326859	Valid picture #4 (Matches MD5)
3	00000002.jpg	afd55222024a4e22f7f5a 3a665320763	175630	Valid picture #3 (Matches MD5)
4	00000003.jpg	7fc3954d980a643e9eafd 62e053cb075	1681986	Corrupted picture #10
5	00000004.jpg	de5d8315333993137171 9f4e5c924eba	26081	Valid picture #2 (Matches MD5)
6	00000005.jpg	35c9da622659465956cf 2d210c89bf07	271181	Valid picture #8
7	00000006.jpg	936d202fbedecbe64b42 c5f3d03233e5	110373	Valid picture #9
8	00000007.doc	4bf26623e510df480200 56fc0ec6d665	154624	Word doc containing picture #9
9	00000008.doc	3c17730f7e132f751015 d025b0f20ef0	3696640	Invalid Word Document

Table 21. Foremost (0.69) results from NTFS sample image

Results from Foremost 1.0 are provided in Table 22 below. Note the only file that could not be fully extracted is the one located in the alternate data stream as the data portion of the file is not contiguous. The fact that 7 of 11 files matched their original md5 hash shows the precision that tailored extraction heuristics offers the disc carving arena. This is an obvious improvement over the 4 files matched by Foremost 0.69 matched and the single md5 matched by ILOOK.

Num	Name	MD5	Size	Note
1	00530.jpg	75b8d00568815a36c380 9b46fc84ba6d	274260	Valid picture #1 (Matches MD5)
2	01066.jpg	0c452c5800fcfa7c66027 ae89c4f068a	326859	Valid picture #4 (Matches MD5)
3	01705.jpg	afd55222024a4e22f7f5a 3a665320763	175630	Valid picture #3 (Matches MD5)
4	06688.jpg	7fc3954d980a643e9eafd 62e053cb075	1681986	Corrupted picture #10
5	10056.jpg	de5d8315333993137171 9f4e5c924eba	26081	Valid picture #2 (Matches MD5)
6	10405.gz	d4f8cf643141f0c2911c5 39750e18ef2	207272	tar ball containing picture #7 (Matches MD5)
7	10810.zip	d41b56e0a9f84eb2825e 73c24cedd963	335371	Archive containing picture #5 (Matches MD5)
8	11466.zip	73c3029066aee9416a5a eb98a5c55321	294124	Archive containing picture #6 (Matches MD5)
9	12044.jpg	35c9da622659465956cf 2d210c89bf07	271181	Valid picture #8
10	12574.doc	0572c54544b657477eeb b25df6cef12c	132096	Word doc containing picture #9
11	12583.jpg	936d202fbedecbe64b42 c5f3d03233e5	110373	Valid picture #9

Table 22. Foremost (1.0) results from NTFS sample image

C. FAT32

The FAT32 image used is a custom 62MB image I created using the mkfs tool. It was created to display the inadequacies of the current data carving tools and to show how some simple methods can be used to improve upon them. This image can be downloaded from Brian Carriers forensic testing site at http://dftt.sourceforge.net/. The drive was also overwritten with zero's to ensure that no other data would be present other than the test images. The first block of the image is also destroyed so that it cannot be mounted. Listed below in Table 23 are the files contained in the image along with their associated attributes and description. These provide the data that can be used for comparison among the different programs.

Num	Name	MD5	Size	Note
1	2003_document.doc	e72f388b36f9370f1969 6b164c308482	19968	A Valid DOC file
2	enterprise.wav	7629b89adade055f6783 dc1773274215	318895	A valid WAV file
3	haxor2.jpg	84e1dceac2eb127fef5b fdcb0eae324b	24367	An invalid JPEG with only 1 header byte corrupted.
4	holly.xls	7917baf0219645afef8b 381570c41211	23040	A valid XLS file
5	lin_1.2.pdf	e026ec863410725ba1f5 765a1874800d	1399508	A linearized PDF
6	nlin_14.pdf	5b3e806e8c9c06a475cd 45bf821af709	122434	A non-linearized PDF
7	paul.jpg	37a49f97ed279832cd4f 7bd002c826a2	29885	A valid jpeg
8	pumpkin.jpg	6c9859e5121ff54d5d62 98f65f0bf3b3	444314	A valid EXIF jpeg
9	shark.jpg	d83428b8742a075b57b0 dc424cd297c4	99298	A valid JPEG
10	sm1.gif	d25fb845e6a41395adae d8bd14db7bf2	5498	A valid GIF
11	surf.mov	5328d2b066f428ea95b2 793849ab97fa	550653	A valid MOV
12	surf.wmv	ff085d0c4d0e0fdc8f34 27db68e26266	1036994	A valid WMV
13	test.ppt	7b74c2c608d92f4bb76c 1d3b6bd1decc	11264	A deleted PPT
14	wword60t.zip	c0be59d49b7ee0fdc492 d2df32f2c6c6	78899	A valid ZIP
15	domopers.wmv	63c0c6986cf0a446cb54 b0ac65a921a5	8037267	A deleted wmv

Table 23. Sample FAT32 test image

The results from Foremost version (0.69) are shown below in Table 24. Notice that version 0.69 extracted 6 out of 14 valid files, but it also generated 5 corrupted files or false positives. Two jpeg images were missed because of a variable JPEG signature (EXIF) that version 0.69 doesn't support. In addition the only reason OLE documents were successfully extracted is because they can contain garbage data at the end of the document hence the large file sizes that 0.69 extracted. This is why 00000010.doc will open successfully however it is over 1000 times as large as the original file size of 11,264 (test.ppt) bytes. The same holds for the 00000002.doc file which was originally only 19,968 bytes in size, but was ballooned to 8,402,944 bytes by 0.69! This method may be satisfactory for small files but this type of extraneous extraction really slows down the program when analyzing larger images. Version 0.69 also extracted a JPEG that had been purposely corrupted to illustrate such inadequacies. Methods such as these rely on the examiner to determine what files are readable/corrupted or not. In addition Foremost 0.69 cannot make a distinction between Word Documents and other OLE files thus it names any OLE file as it were a word document.

Num	Name	MD5	Size	Note
1	00000000.jpg	84e1dceac2eb127fef5bf dcb0eae324b	24367	Corrupted JPEG
2	00000001.jpg	37a49f97ed279832cd4f 7bd002c826a2	29885	Valid JPEG (paul.jpg) (matches md5)
3	00000002.doc	a4aa85035d929bc5a9bb b2f2b5e1f2d0	8402944	Valid DOC (2003_document.doc)
4	00000003.doc	32b48b4fd63d7ebae885 f31cc64914f2	3719168	Valid XLS (stats.xls)
5	00000004.pdf	1c4f8da888e2a032afdf7 7b2157d3074	5000000	Invalid PDF
6	00000005.gif	a80122dbb804f919b1fb 688acf57782f	63677	Valid GIF (sm1.gif)
7	00000006.jpg	7e0b420a2ea2258b8743 b9abef7c6946	3051	Invalid JPG
8	00000007.jpg	635ed8b379942f6cda5e 6c809c52f8a1	2655	Thumbnail of shark.jpg
9	00000008.jpg	635ed8b379942f6cda5e 6c809c52f8a1	2655	Thumbnail of shark.jpg
10	00000009.mov	b8c798ce4204018e35f8 e7e2e749a73d	4000000	Invalid MOV
11	00000010.doc	bc20b8af9754d9b0d615 88fdd9fdba0c	12500000	Valid PPT (test.ppt)

Table 24. Foremost (0.69) results from FAT32 sample image

The results from Foremost (1.0) are included in Table 25 below. Version 1.0 successfully recovered all 14 valid files and ignores the corrupted JPEG file (haxor2.jpg). This method also reduces the amount of redundant processing that version 0.69 does and speeds up the processing exponentially. 10 out of 14 files match their original md5sum and the rest are no more than a few sectors off from their original size. This adds weight in a forensic context as the evidence is more precise than version 0.69 which only matches 1 out of 14 md5 hashes.

Num	Name	MD5	Size	Note
1	19717.jpg	37a49f97ed279832cd4f 7bd002c826a2	29885	Valid JPEG (paul.jpg) (Matches md5)
2	19777.jpg	6c9859e5121ff54d5d62 98f65f0bf3b3	444314	Valid JPEG (pumpkin.jpg) (Matches md5)
3	20645.jpg	d83428b8742a075b57b0 dc424cd297c4	99298	Valid JPEG (shark.jpg) (Matches md5)
4	20841.gif	d25fb845e6a41395adae d8bd14db7bf2	5498	Valid GIF (sm1.gif) (Matches md5)
5	321.wmv	63c0c6986cf0a446cb54 b0ac65a921a5	8037267	Valid WMV (domopers.wmv) (Matches md5)
6	21929.wmv	ff085d0c4d0e0fdc8f342 7db68e26266	1036994	Valid WMV (surf.wmv) (Matches md5)
7	20853.mov	5328d2b066f428ea95b2 793849ab97fa	550653	Valid MOV (surf.mov) (Matches md5)
8	16021.wav	4020b55670015ee50672 260efd138aff	318886	Valid WAV (enterprise.wav)
9	281.doc	5ae5cd40c3d07d5df554 b2030a001ebd	20992	Valid Word Document (2003_document.doc)
10	16693.xls	a9bba638866a7f5ba4ba db727a1628c9	25088	Valid XLS (stats.xls)
11	23957.ppt	da30aae8b23194e11302 20d47ceddfed	13312	Valid PPT (test.ppt)
12	23981.zip	c0be59d49b7ee0fdc492 d2df32f2c6c6	78899	A valid ZIP file(wword60t.zip) (Matches md5)
13	16741.pdf	e026ec863410725ba1f5 765a1874800d	1399508	A valid PDF (lin_1.2.pdf) (Matches md5)
14	19477.pdf	5b3e806e8c9c06a475cd 45bf821af709	122434	A valid PDF (nlin_14.pdf) (Matches md5)

Table 25. Foremost (1.0) results from FAT32 sample image

D. EXT2/EXT3

The EXT2 image studied is a 62MB image I created from a USB thumb drive. This image along with its hash is available via the internet at http://dftt.sourceforge.net/. The drive was formatted using the mkfs program so that indirect block detection could be evaluated. After the image was constructed the meta data pertaining to mounting the image was corrupted to ensure strict carving methods would be used to extract data. The default block size chosen by the mkfs program is 1024, therefore Foremost 1.0 should detect single indirect blocks and remove them. Many of the files included in the image are larger than 12,168 bytes, thus they require at least a single indirect block.

Num	Name	MD5	Size	Note	Blocks (bs=1024)
1	haxor2.bmp	f9633fe6b9ef2a0a5edd6de70d22c0f5	163878	A deleted BMP	(0-11):2581-2592, (IND):2593, (12-160):2594-2742
2	jimmy.doc	2f3f914dd74819df42d1d941c7275c16	12800	A deleted DOC	(0-11):2743-2754, (IND):2755, (12):2756
3	jn.jpg	270a0a913fa9603db8121fdf78d63aca	28949	A valid JPG	(0-11):2757-2768, (IND):2769, (12-28):2770-2786
4	lin_test.pdf	1c64456776075d1f0a662e1f6c09e340	26618	A valid PDF	(0-11):2787-2798, (IND):2799, (12-25):2800-2813
5	main_dive.jpg	937846adb96773ee25fcb34821230976	8463	A valid jpeg	(0-8):2814-2822
6	n_lin_ss.pdf	97be95ed3e710b63bc75e5c0775062d9	734652	A valid pdf	(0-11):2823-2834, (IND):2835, (12-267):2836-3091, (DIND):3092, (IND):3093, (268-523):3094-3349, (IND):3350, (524-717):3351-3544
7	blogo.gif	5e10b2176016885a85bffc074a142524	18663	A valid gif	(0-11):2561-2572, (IND):2573, (12-18):2574-2580
8	sherry.jpg	3834e72d2ee266ccfb9733d716b89f2b	133249	A valid JPEG	(0-11):3545-3556, (IND):3557, (12-130):3558-3676
9	stats.xls	6351df9c1543c41c3df8eea63e06a219	15360	A valid XLS	(0-11):3677-3688, (IND):3689, (12-14):3690-3692
10	test.ppt	99941c129cc8cfbadc15c55086982efc	17408	A valid PPT	(0-11):3693-3704, (IND):3705, (12-16):3706-3710

Table 26. Sample EXT2 Image

The results from Foremost version (0.69) are shown in Table 27 below. The only file that was successfully extracted by version 0.69 was smaller than 12,168 bytes and thus didn't include any indirect blocks. Only one of the file matched its original

MD5SUM hash. 4 of the 9 files extracted are at least partially viewable. Most notably 00000001.jpg and 00000000.gif still contain their indirect blocks and thus the latter halves of the images are not-viewable. Also note that only the thumbnail of sherry.jpg was extracted because version 0.69 doesn't adequately recognize an EXIF JPEG. This example demonstrates the inadequacies of Foremost (0.69) in analyzing a UNIX file-system.

Num	Name	MD5	Size	Note
1	00000000.gif	c36a312216225baff5b0 8bba5dab00e6	19687	Partially corrupted GIF (blogo.gif)
2	00000001.jpg	305b1d7092fe993f35dd 3aa4bc49f283	29973	Partially corrupted JPEG (jn.jpg)
3	00000002.jpg	937846adb96773ee25fc b34821230976	8463	A valid JPEG(main_dive.jpg) (matches md5)
4	00000003.jpg	4b4a4fe7392157d8f2bf4 5b3a0238309	7043	Corrupted JPEG
5	00000004.jpg	d21f50c6f46d8db20dbf2 34284b70f8f	4905	Thumbnail of (sherry.jpg)
6	00000005.doc	21012bdaf757ce6c68dfc 3fb4184c199	956416	An invalid DOC
7	00000006.doc	93ee406edf4e68f91a5a9 cdddc28132b	16384	An invalid XLS
8	00000007.doc	524124dbbcc6da91d677 f851921f2366	12500000	An invalid PPT
9	00000007.pdf	05739489a3fc08858557 acf69b192497	5000000	An invalid PDF

Table 27. Foremost (0.69) results from EXT2 sample image

The results from Foremost (1.0) are included below in Table 28. The only real problem version 1.0 ran into is the fact that n_lin_ss.pdf requires a double indirect block which is not supported in this version. This extension is left as future work. 5 out of 10 MD5SUMS matched and all of the files were at least partially viewable as compared to the previous version where only 4 files were even partially discernable and only 1 MD5SUM matched. Also note that over half of the files were not readable thus this causes the examiner for time to manually extract files.

Num	Name	MD5	Size	Note
1	5514.jpg	270a0a913fa9603db812 1fdf78d63aca	28949	Valid JPEG (jn.jpg) (matches md5)
2	5626.jpg	937846adb96773ee25fc b34821230976	8463	Valid JPEG (main_dive.jpg) (matches md5)
3	7088.jpg	432a6017f18abca995e0 e708a1ff18b6	133249	Valid JPEG (sherry.jpg) (matches md5)
4	5122.gif	5e10b2176016885a85bf fc074a142524	18663	Valid GIF (blogo.gif) (matches md5)
5	5160.bmp	f9633fe6b9ef2a0a5edd6 de70d22c0f5	163878	Valid BMP (haxor2.bmp) (matches md5)
6	5482.doc	b930aaee0c478ad69bc8 63349b7b899d	14848	Valid DOC (jimmy.doc)
7	7344.xls	dad72c2effb3aa93c3845 fbc05de6622	17408	Valid XLS (stats.xls)
8	7374.ppt	068114007cde9e94e5aa 4236f0c79e65	19456	A valid PPT (test.ppt)
9	5566.pdf	1c64456776075d1f0a66 2e1f6c09e340	26618	A valid PDF (lin_test.pdf) (matches md5)
10	5636.pdf	2d4831f8a0c70844a126 d961fca3792b	738748	Partially corrupted PDF(n_lin_ss.pdf)

Table 28. Foremost (1.0) results from EXT2 sample image

This test case shows the case where adding 50 lines of code to a program can dramatically increase the extraction functionality of a given tool. Granted that indirect block detection is not an exact science, but it does provide more useful data when extracting files from a UNIX file system. Sample source code is provided for indirect block detection in Appendix A.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION

A. SUMMARY

With some study of file format specifications and reverse engineering of propriety formats, existing disc carving tools can be vastly improved. In addition through comparison with closed source products this paper has shown that open source tools can perform just as well, if not better than commercial forensic suites. The methods outlined in this paper can provide a file system independent program that can take advantage of file system specific information such as indirect block detection but not use them as a crutch.

Experimental results provided in this paper as well as those performed on real world machines have shown the usefulness of developing more sophisticated disc carving algorithms. As file systems and file formats become more complex so must the intelligence of these algorithms in order to preserve forensic integrity and utility.

The current implementation of the algorithms described in chapter III can be viewed in the CVS repository of Foremost at http://sourceforge.net/. At the time of this paper version 1.0 is in its testing phase, once completed it will be made available at http://foremost.sourceforge.net/.

B. PROBLEMS

The code, as provided in Foremost version 0.69, is somewhat platform dependent and needs to be rewritten to encourage portability/modularity to at least other UNIX platforms if not Windows. The main reason that this code has not been incorporated into Sleuthkit is the fact that it is very Linux dependent and cannot be easily ported to Solaris or BSD. Jesse Kornblum (The original author) is rewriting the entire program for this reason. Once this is complete, the work described in this thesis will be incorporated into the new version. The inclusion of Foremost into Sleuthkit will give added weight to the forensic suite and hopefully increase its popularity within organizations that can not afford expensive Windows based products, or wish to make use of open source solutions.

One of the main problems faced in developing a tool such as Foremost is the fact that the memory of the machine used for analysis is finite. This problem manifests itself when attempting to extract files that span our "chunk" size. The default chunk size used by Foremost is 100MB, thus large files are analyzed 100MB at a time. What are the best methods to "bridge the gap" between chunks while analyzing an image? The easy answer is to just re-read from the disk every time we find a file near the edge of a chunk, however, disk reads are inefficient and should be minimized. Foremost 1.0 uses the "max file size" approach to deal with this problem. A look ahead can be performed to meet this size. A simpler approach is to use very high end machines with large amounts of memory. The problem would be reduced as the amount of available memory grows.

C. FUTURE WORK

The creation of a standard library of file specific extraction methods so all forensic tools can have access to the same robust algorithms to carve data would be a significant capability for the forensics community. This would allow tools to focus on other areas of forensic research while having a powerful set of methods to detect and extract given file formats.

In addition to file recognition, block recognition poses a more complicated problem. As a file-system becomes more fragmented this will be a technology that must be employed in an attempt to continue the usefulness of disc carving. This is especially relevant when taking RAM images into account as paging leaves files seemingly scattered across the image. If these blocks could be detected and rebuilt to be fed to an extraction algorithm that can detect valid file formats this would greatly improve live forensic capabilities.

Improvements of OLE and GZIP extraction methods require more study than is covered in the scope of this paper. The available documentation of OLE file structure is limited. Existing methods are in place via the API and programs provided by the Chicago Project. However, these methods do not provide adequate means to determine the actual file size. OLE documents are notorious for their garbage data and wasted space. There are tools available to read and write to this "garbage" area of the file. More research and reverse engineering are needed to be able to track this space so it can be accounted for when determining file sizes. Our experimentation using the algorithm described previously shows that one can usually determine file sizes within a block of the actual file end. This is adequate if the goal is to read the document but doesn't provide accurate results in terms of forensics as the extracted data is not identical to the actual file on this disk.

GZIP file detection lacks functionality without a GZIP decompression algorithm, as described previously. Such a method incorporated in Foremost would allow for more accurate extractions as well as the inflation of archived files on the fly.

Single indirect block detection provides a simple and useful tool to aide the forensic analysis of UNIX file-systems. However, being able to provide additional logic to rebuild files based on their single as well as double and triple indirect blocks poses a more challenging problem not addressed in this paper. Such functionality would allow better analysis of file-systems which employ smaller block sizes thus requiring more indirect blocks. In addition large multimedia files and documents could be extracted more efficiently.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. SOURCE CODE

This appendix includes all files which were modified in the development of Foremost 1.0. This version is current as of 3/09/05: please go to http://foremost.sourceforge.net to get the latest copy. The main intelligence of Foremost comes from the extract.c file where all the extraction functions are fully defined.

A. EXTRACT.C

```
/* extract.c
 * Copyright (c) 2005, Nick Mikus
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the Free
 * Software Foundation; either version 2 of the License, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along with
 * this program; if not, write to the Free Software Foundation, Inc., 59 Temple
 * Place, Suite 330, Boston, MA 02111-1307 USA
 *
 * This file contains the file specific functions used to extract
 * data from an image.
 *
 * Each has a similar structure
 * f_state *s:  state of the program.
 * c_offset:       offset that the header was recorded within the current chunk
 * foundat:        The location the header was "foundat"
 * buflen:         How much buffer is left until the end of the current chunk
 * needle:         Search specification
 * f_offset:       Offset that the current chunk is located within the file
 */



#include "main.h"
#include "extract.h"
#include "ole.h"
extern char buffer[OUR_BLK_SIZE];
extern int verbose;
extern int dir_count;
extern int block_list[OUR_BLK_SIZE / sizeof (int)];
extern int *FAT;
extern char *extract_name;
extern int extract;
extern int FATblk;
extern int highblk;


/********************************************************************************
 *Function: extractZIP
 *Description: Given that we have a ZIP header jump through the file headers
    until we reach the EOF.
*Return: A pointer to where the EOF of the ZIP is in the current buffer
```

```
*******************************************************************************/
char* extractZIP(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec* needle,unsigned long long f_offset)
{
    char* currentpos=NULL;
    char* buf=foundat;
    unsigned short comment_length=0;
    char* extractbuf = NULL;
    struct zipLocalFileHeader localFH;
    int bytes_to_search=50*KILOBYTE;
    unsigned long long file_size=0;
    while(1) /*Jump through each local file header until the central directory structure
is reached, much faster than searching */
    {
        if(foundat[2]=='\x03' && foundat[3]=='\x04') /*Verfiy we are looking at a local
file header*/
        {
            localFH.compressed=htoi(&foundat[18],LITTLE_ENDIAN);
            localFH.filename_length=htos(&foundat[26],LITTLE_ENDIAN);
            localFH.extra_length=htos(&foundat[28],LITTLE_ENDIAN);

              /* Sanity checking*/
            if(localFH.compressed > needle->max_len) return foundat+needle->header_len;

            if(localFH.filename_length > 100) return foundat+needle->header_len;

              /*Check if we should grab more from the disk*/
            if(localFH.compressed+30 > buflen-(foundat-buf))
            {
                   return NULL;/*Go back grab more and try again*/
            }
            foundat+=localFH.compressed;
            foundat+=30; /*Size of the local file header data structure*/
            foundat+=localFH.filename_length;
            foundat+=localFH.extra_length;
          #ifdef DEBUG
          printf("localFH.compressed:=%d\n",localFH.compressed);
          #endif
        }
        else
        {
            break;
        }
    }
    bytes_to_search=(foundat-buf);
    if(buflen-(foundat-buf) < bytes_to_search)
    {
        bytes_to_search=buflen-(foundat-buf);
    }


    currentpos=foundat;
#ifdef DEBUG
    printf("Search for the footer bytes_to_search:=%d
buflen:=%lld\n",bytes_to_search,buflen);
#endif

    foundat=  bm_search(needle->footer,needle->footer_len,foundat,bytes_to_search,needle-
>footer_bm_table,needle->case_sen,SEARCHTYPE_FORWARD);
#ifdef DEBUG
    printf("Search complete \n");
#endif


    if(foundat) /*Found the end of the central directory structure, determine the exact
length and extract*/
    {
          /*Jump to the comment length field*/
#ifdef DEBUG
    printf("distance searched:=%d \n",foundat-currentpos);
#endif
```

```
          if(buflen-(foundat-buf) > 20)
          {
                    foundat+=20;
         }
          else
          {
                    return NULL;
          }
          comment_length=htos(foundat,LITTLE_ENDIAN);
        foundat+=comment_length+1;
          file_size = (foundat-buf);
#ifdef DEBUG
    printf("File size %lld\n",file_size);
#endif
        extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
        memcpy(extractbuf,buf,file_size);
        writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);
          free(extractbuf);
        return foundat;
    }
    if(bytes_to_search > buflen-(currentpos-buf)) return NULL;


#ifdef DEBUG
    printf("I give up \n");
#endif
    return currentpos;
}


/*********************************************************************************
 *Function: extractPDF
 *Description: Given that we have a PDF header check if it is Linearized, if so
    grab the file size and we are done, else search for the %%EOF
*Return: A pointer to where the EOF of the PDF is in the current buffer
**********************************************************************************/
char* extractPDF(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec* needle,unsigned long long f_offset)
{
    char* currentpos=NULL;
    char* buf=foundat;
    char* extractbuf = NULL;
    unsigned char* tempsize;
    unsigned long int size=0;
    int file_size=0;
    char* header=foundat;
    int bytes_to_search=0;

    foundat+=needle->header_len;/* Jump Past the %PDF HEADER */
    currentpos=foundat;

     /*Determine when we have searched enough*/
    if(buflen >= needle->max_len)
    {
        bytes_to_search=needle->max_len;
    }
    else
    {
        bytes_to_search=buflen;
    }

    /*Check if the buffer is less than 100 bytes, if so search what we have*/
    if(buflen < 512) return NULL;
    else
    {
        currentpos=foundat;
/*Check for .obj in the first 100 bytes*/
        foundat=  bm_search(needle->markerlist[1].value,needle-
>markerlist[1].len,foundat,100,needle->markerlist[1].marker_bm_table,needle-
>case_sen,SEARCHTYPE_FORWARD);
        if(!foundat)
```

```
        {
#ifdef DEBUG
                   printf("no obj found\n");
#endif
            return currentpos+100;
        }
        foundat=currentpos;

/*Search for "./L " to see if the file is linearized*/

         foundat=  bm_search(needle->markerlist[2].value,needle-
>markerlist[2].len,foundat,512,needle->markerlist[2].marker_bm_table,needle-
>case_sen,SEARCHTYPE_FORWARD);

         if(foundat)
         {
         foundat=  bm_search(needle->markerlist[0].value,needle-
>markerlist[0].len,foundat,512,needle->markerlist[0].marker_bm_table,needle-
>case_sen,SEARCHTYPE_FORWARD);
         }
         else
         {
#ifdef DEBUG
                   printf("not linearized\n");
#endif
         }
    }

    if(foundat) /*The PDF is linearized extract the size and we are done*/
    {

        foundat+=needle->markerlist[0].len;
        tempsize=(char*) malloc(8*sizeof(char));
        tempsize=memcpy(tempsize,foundat,8);
        size=atoi(tempsize);

         free(tempsize);
        if(size <=0 ) return foundat;
         if(size > buflen)
        {
                   if(size > needle->max_len) return foundat;
                   else return NULL;
        }
        header+=size;
        foundat=header;
        foundat-=needle->footer_len;
         /*Jump back 10 bytes and see if we actually have and EOF there*/
        foundat-=10;
        currentpos=foundat;
        foundat=  bm_search(needle->footer,needle->footer_len,foundat,needle-
>footer_len+9,needle->footer_bm_table,needle->case_sen,SEARCHTYPE_FORWARD);
        if(foundat)/*There is an valid EOF at the end, Write to disk*/
        {

            foundat+=needle->footer_len+1;
            file_size = (foundat-buf);

            extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
             memcpy(extractbuf,buf,file_size);
            writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);
             free(extractbuf);
             return foundat;
        }
         return NULL;

    }
    else /*Search for Linearized PDF failed, just look for %%EOF */
    {
#ifdef DEBUG
        printf("   Linearized search failed, searching %d bytes,
buflen:=%lld\n",bytes_to_search,buflen-(header-buf));
```

```
#endif
         foundat=currentpos;
        foundat=  bm_search(needle->footer,needle-
>footer_len,foundat,bytes_to_search,needle->footer_bm_table,needle-
>case_sen,SEARCHTYPE_FORWARD);

         if(foundat) /*Write the non-linearized PDF to disk*/
         {

           foundat+=needle->footer_len+1;
           file_size = (foundat-buf);

           extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
            memcpy(extractbuf,buf,file_size);
           writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);
            free(extractbuf);
            return foundat;

         }
         return NULL;
    }


}

/*****************************************************************************
 *Function: extractCPP
 *Description: Use keywords to attempt to find C/C++ source code
*Return: A pointer to where the EOF of the CPP file is in the current buffer
*****************************************************************************/
char* extractCPP(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec* needle,unsigned long long f_offset)
{

    char* header=foundat;
    char* buf=foundat;
    char* extractbuf = NULL;
    int end=0;
    int start=0;
    int i=0;
    int marker_score=0;
    int ok=FALSE;
    int file_size=0;
    char* footer=NULL;

    /*Search for a " or a < within 20 bytes of a #include statement*/
    for(i=0;i<20;i++)
    {
        if(foundat[i]=='\x22' || foundat[i]=='\x3C')
        {
            ok=TRUE;
        }
    }

    if(!ok) return foundat+needle->header_len;
    /*Keep running through the buffer until an non printable character is reached*/
    while(isprint(foundat[end]) || foundat[end]=='\x0a' || foundat[end]=='\x09')
    {
        end++;
    }
    foundat+=end-1;
    footer=foundat;

    if(end < 50) return foundat;

    /*Now lets go the other way and grab all those comments at the begining of the file*/
    while(isprint(buf[start]) || buf[start]=='\x0a' || buf[start]=='\x09')
    {
        start--;
    }
```

```
    header=&buf[start+1];
    file_size=(footer-header);

    foundat=header;

    /*Now we have an ascii file to look for keywords in*/
    foundat=  bm_search(needle->footer,needle->footer_len,header, file_size,needle-
>footer_bm_table,FALSE,SEARCHTYPE_FORWARD);
    if(foundat) marker_score+=1;

    foundat=header;
    foundat=  bm_search(needle->markerlist[0].value,needle->markerlist[0].len,header,
file_size,needle->markerlist[0].marker_bm_table,1,SEARCHTYPE_FORWARD);
    if(foundat) marker_score+=1;


    if(marker_score == 0) return foundat;

    if(foundat)
    {
         extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
         memcpy(extractbuf,header,file_size);
         writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset+start+1);
         free(extractbuf);
         return footer;

    }
    return NULL;
}


/********************************************************************************
 *Function: extractHTM
 *Description: Given that we have a HTM header
    search for the file EOF and check that the bytes areound the header are ascii
*Return: A pointer to where the EOF of the HTM is in the current buffer
**********************************************************************************/
char* extractHTM(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec* needle,unsigned long long f_offset)
{
    char* buf=foundat;
    char* extractbuf = NULL;
    char* currentpos=NULL;

    int bytes_to_search=0;
    int i=0;
    int file_size=0;

    /*Jump past the <HTML tag*/
    foundat+=needle->header_len;

    /*Check the first 16 bytes to see if they are ASCII*/
    for(i=0;i<16;i++)
    {
        if(!isprint(foundat[i]) && foundat[i]!='\x0a' && foundat[i]!='\x09')
        {
            return foundat+16;
        }
    }

    /*Determine if the buffer is large enough to encompass a reasonable search*/
    if(buflen < needle->max_len)
    {
        bytes_to_search=buflen-(foundat-buf);
    }
    else
    {
        bytes_to_search=needle->max_len;
    }

    /*Store the current position and search for the HTML> tag*/
```

```
    currentpos=foundat;
    foundat=   bm_search(needle->footer,needle-
>footer_len,foundat,bytes_to_search,needle->footer_bm_table,needle-
>case_sen,SEARCHTYPE_FORWARD);
    if(foundat)//Found the footer, write to disk
    {
          file_size = (foundat-buf)+needle->footer_len;
          extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
          memcpy(extractbuf,buf,file_size);
          writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);
          free(extractbuf);
          foundat+=needle->footer_len;
          return foundat;

    }
    else
    {
        return NULL;
    }

}


/********************************************************************************
 *Function: validOLEheader
 *Description: run various tests aginst an OLE-HEADER to determine whether or not
          it is valid.
*Return: TRUE/FALSE
*********************************************************************************/
int validOLEheader(struct OLE_HDR *h)
{

    if(htos((char*) &h->reserved,FOREMOST_LITTLE_ENDIAN) !=0 || htoi((char*) &h-
>reserved1,FOREMOST_LITTLE_ENDIAN)!=0 || htoi((char*) &h-
>reserved2,FOREMOST_LITTLE_ENDIAN)!=0)
    {
        return FALSE;
    }

    /*The minimum sector shift is usually 2^6(64) and the uSectorShift is 2^9(512))*/
    if(htos((char*) &h->uMiniSectorShift,FOREMOST_LITTLE_ENDIAN)!=6 || htos((char*) &h-
>uSectorShift,FOREMOST_LITTLE_ENDIAN)!=9 || htoi((char*) &h-
>dir_flag,FOREMOST_LITTLE_ENDIAN) < 0)
    {
        return FALSE;
    }
    /*Sanity Checking*/
    if(htoi((char*) &h->num_FAT_blocks,FOREMOST_LITTLE_ENDIAN) <= 0 || htoi((char*) &h-
>num_FAT_blocks,FOREMOST_LITTLE_ENDIAN) > 100)
    {
        return FALSE;
    }
    if(htoi((char*) &h->num_extra_FAT_blocks,FOREMOST_LITTLE_ENDIAN) < 0 || htoi((char*)
&h->num_extra_FAT_blocks,FOREMOST_LITTLE_ENDIAN) > 100)
    {
        return FALSE;
    }
    return TRUE;

}

/********************************************************************************
 *Function:checkOleName
 *Description: Determine what type of file is stored in the OLE format based on the
          names of DIRENT in the FAT table.
*Return: A char* consisting of the suffix of the appropriate file.
*********************************************************************************/
char* checkOleName(char* name)
{
    if(strstr(name,"WordDocument"))
    {
```

```
        return "doc";
    }
    else if(strstr(name,"Worksheet") || strstr(name,"Book") || strstr(name,"Workbook"))
    {
        return "xls";
    }
    else if(strstr(name,"Power"))
    {
        return "ppt";
    }
    else if(strstr(name,"Access") || strstr(name,"AccessObjSiteData"))
    {
        return "mbd";
    }
    else if(strstr(name,"Visio"))
    {
        return "vis";
    }
    else if(strstr(name,"Sfx"))
    {
        return "sdw";
    }
    else
    {
        return NULL;
    }

    return NULL;

}

int adjustBS(int size,int bs)
{
          int rem=(size%bs);

          if(rem==0)
          {

                    return size;
          }
#ifdef DEBUG
          printf("\tnew size:=%d\n",size+(bs-rem));
#endif
          return (size+(bs-rem));

}


/********************************************************************************
 *Function: extractOLE
 *Description: Given that we have a OLE header, jump through the OLE structure and
    determine what type of file it is.
*Return: A pointer to where the EOF of the OLE is in the current buffer
********************************************************************************/
char* extractOLE(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec* needle,unsigned long long f_offset,char* type)
{
    char* buf=foundat;
    char* extractbuf = NULL;
    char* temp=NULL;
    char*  suffix="ole";
    int totalsize=0;
    int extrasize=0;
    int oldblk=0;
    int i, j;
    int size=0;
    int blknum=0;
    int validblk=512;
    int file_size=0;
    int num_extra_FAT_blocks=0;
    char* htoi_c=NULL;
```

```
    int extra_dir_blocks=0;
    int num_FAT_blocks=0;
    int next_FAT_block=0;
    char *p;
    int fib=1024;
    struct OLE_HDR *h = NULL;

    int result=0;
    int highblock=0;
    unsigned long  miniSectorCutoff=0;
    unsigned long  csectMiniFat=0;

     /*Deal with globals defined in the OLE API, ugly*/
    if(dirlist!=NULL)  free(dirlist);
    if(FAT!=NULL)      free (FAT);
    initOLE();

    if(buflen < validblk) validblk=buflen;
    h = (struct OLE_HDR*) foundat; /*cast the header block to point at foundat*/
#ifdef DEBUG
    dump_header(h);
#endif
    num_FAT_blocks=htoi((char*) &h->num_FAT_blocks,FOREMOST_LITTLE_ENDIAN);

    if(!validOLEheader(h)) return (buf+validblk);

    miniSectorCutoff=htoi((char*) &h->miniSectorCutoff,FOREMOST_LITTLE_ENDIAN);
    csectMiniFat=htoi((char*) &h->csectMiniFat,FOREMOST_LITTLE_ENDIAN);
    next_FAT_block=htoi((char*) &h->FAT_next_block,FOREMOST_LITTLE_ENDIAN);
    num_extra_FAT_blocks=htoi((char*) &h->num_extra_FAT_blocks,FOREMOST_LITTLE_ENDIAN);

    FAT = (int *) Malloc (OUR_BLK_SIZE * (num_FAT_blocks + 1));
    p = (char *) FAT;
    memcpy (p, &h[1], OUR_BLK_SIZE - FAT_START);
    if (next_FAT_block > 0)
    {
        p += (OUR_BLK_SIZE - FAT_START);
        blknum = next_FAT_block;
        for (i = 0; i <  num_extra_FAT_blocks; i++)
        {
              if(!get_block (buf, blknum,p, buflen)) return buf+validblk;
            validblk=(blknum+1)*OUR_BLK_SIZE;
            p += OUR_BLK_SIZE - sizeof (int);
            blknum = htoi(p,FOREMOST_LITTLE_ENDIAN);
        }
    }


    blknum = htoi((char*) &h->root_start_block,FOREMOST_LITTLE_ENDIAN);
    highblock=htoi((char*) &h->dir_flag,FOREMOST_LITTLE_ENDIAN);
#ifdef DEBUG
          printf("getting dir block\n");
#endif
    //if(!get_dir_block (buf, blknum, buflen)) return buf+validblk;

    if(!get_block (buf, blknum,buffer, buflen))return buf+validblk;/*GET DIR BLOCK*/
#ifdef DEBUG
          printf("done getting dir block\n");
#endif
    validblk=(blknum+1)*OUR_BLK_SIZE;
    while (blknum != END_OF_CHAIN)
    {
#ifdef DEBUG
          printf("finding dir info extra_dir_blks:=%d\n",extra_dir_blocks);
#endif
          if(extra_dir_blocks > 300) return  buf+validblk;

/**PROBLEMA**/
#ifdef DEBUG
          printf("***blknum:=%d FATblk:=%d\n",blknum,FATblk);
#endif
```

```
        oldblk=blknum;
         htoi_c=(char *) &FAT[blknum / (OUR_BLK_SIZE / sizeof (int))];

        FATblk = htoi(htoi_c,FOREMOST_LITTLE_ENDIAN);
#ifdef DEBUG
         printf("***blknum:=%d FATblk:=%d\n",blknum,FATblk);
#endif

        if(!get_FAT_block (buf, blknum, block_list,buflen)) return buf+validblk;
        blknum = htoi((char *) &block_list[blknum % 128],FOREMOST_LITTLE_ENDIAN);
#ifdef DEBUG
         printf("**blknum:=%d FATblk:=%d\n",blknum,FATblk);
#endif
        if (blknum == END_OF_CHAIN || oldblk==blknum)
         {
#ifdef DEBUG
            printf("EOC\n");
#endif
                break;
        }
         extra_dir_blocks++;
        result=get_dir_block (buf, blknum,buflen);
        if (result==SHORT_BLOCK)
        {
#ifdef DEBUG
            printf("SHORT BLK\n");
#endif
            break;
        }
        else if(!result) return buf+validblk;

    }
#ifdef DEBUG
            printf("DONE WITH WHILE\n");
#endif
    blknum = htoi((char*) &h->root_start_block,FOREMOST_LITTLE_ENDIAN);
    size = OUR_BLK_SIZE * (extra_dir_blocks + 1);
    dirlist = (struct DIRECTORY *) Malloc (size);
    memset (dirlist, 0, size);

    if(!get_block (buf, blknum,buffer, buflen))return buf+validblk;/*GET DIR BLOCK*/

    if(!get_dir_info (buffer))
    {
         return foundat+validblk;
    }

    for (i = 0; i < extra_dir_blocks; i++)
    {
        if(!get_FAT_block (buf, blknum, block_list,buflen)) return buf+validblk;
        blknum = htoi((char *) &block_list[blknum % 128],FOREMOST_LITTLE_ENDIAN);
        if (blknum == END_OF_CHAIN)
            break;
#ifdef DEBUG
        printf("getting dir blk blknum=%d\n",blknum);
#endif
        if(!get_block (buf, blknum,buffer, buflen))return buf+validblk;/*GET DIR BLOCK*/
         if(!get_dir_info (buffer))
         {
                return buf+validblk;
         }
    }
#ifdef DEBUG
         printf("dir count is %d\n",i);
#endif
    for (dl = dirlist, i = 0; i < dir_count; i++, dl++)
    {
        memset (buffer, ' ', 75);
        j = htoi((char*) &dl->level,FOREMOST_LITTLE_ENDIAN)*4;
        sprintf (&buffer[j], "%-s", dl->name);
        j = strlen (buffer);
```

```
        if(dl->name[0]=='@') return foundat+validblk;
        if (dl->type == STREAM)
        {
            buffer[j] = ' ';
            sprintf (&buffer[60], "%8d\n", dl->size);

              if(temp==NULL) /*check if we have alread defined the type*/
              {
                    temp=checkOleName(dl->name);
                    if(temp) suffix=temp;
              }
            if(dl->size > miniSectorCutoff)
              {
                    totalsize+=adjustBS(dl->size,512);
            }
              else
              {
                    totalsize+=adjustBS(dl->size,64);
              }

#ifdef DEBUG
            fprintf (stdout, buffer);
#endif
        }
        else
        {
            sprintf (&buffer[j], "\n");
#ifdef DEBUG
              printf("\tnot stream data \n");
              fprintf (stdout, buffer);
#endif

            extrasize+=adjustBS(dl->size,512);

        }
    }


    totalsize+=fib;
#ifdef DEBUG
printf("DIR SIZE:=%d, numFATblks:=%d
MiniFat:=%d\n",adjustBS(((dir_count)*128),512),(num_FAT_blocks*512),adjustBS((64*csectMin
iFat),512));
#endif
    totalsize+=adjustBS(((dir_count)*128),512);
    totalsize+=(num_FAT_blocks*512);
    totalsize+= adjustBS((64*csectMiniFat),512);
     if((highblk+5) > highblock && highblk > 0)
    {
          highblock=highblk+5;
    }
    highblock=highblock*512;

#ifdef DEBUG
          printf("\t highblock:=%d\n",highblock);
#endif
    if(highblock > totalsize)
    {
#ifdef DEBUG
          printf("  Total size:=%d a difference of %lld\n",totalsize,buflen-totalsize);
          printf("  Extra size:=%d \n",extrasize);
        printf("   Highblock is greater than totalsize\n");
#endif
        totalsize=highblock;
    }

   totalsize=adjustBS(totalsize,512);
#ifdef DEBUG
    printf("       Total size:=%d a difference of %lld\n",totalsize,buflen-totalsize);
    printf("       Extra size:=%d \n",extrasize);
```

```
#endif


    if(buflen < totalsize)
    {
#ifdef DEBUG
        printf("   ***Error not enough left in the buffer left:=%lld
needed=%d***\n",buflen, totalsize);
#endif
        totalsize=buflen;
    }



    foundat=buf;
    highblock-=5*512;
    if(highblock > 0 && highblock < buflen)
    {
         foundat+=highblock;
    }
    else
    {
         foundat+=totalsize;
    }
    /*Return to the highest blknum read in the file, that way we don't miss files that
are close*/

    file_size = totalsize;
    extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
    memcpy(extractbuf,buf,file_size);
    if(suffix) needle->suffix=suffix;

    if(!strstr(needle->suffix,type) && type!="all")
    {
         return foundat;
    }
    writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);

    free(extractbuf);
    return foundat;

}


//*****************************************************************************/
int checkMov(char* atom)
{
#ifdef DEBUG
          printf("Atom:= %c%c%c%c\n",atom[0],atom[1],atom[2],atom[3]);
#endif
          if(strncmp(atom,"free",4)==0 || strncmp(atom,"mdat",4)==0 ||
strncmp(atom,"free",4)==0 || strncmp(atom,"wide",4)==0 || strncmp(atom,"PICT",4)==0)
          {
                    return TRUE;
          }
          if(strncmp(atom,"trak",4)==0 || strncmp(atom,"mdat",4)==0 ||
strncmp(atom,"mp3",3)==0 || strncmp(atom,"wide",4)==0 || strncmp(atom,"moov",4)==0)
          {
                    return TRUE;
          }

          return FALSE;
}
/****************************************************************************
 *Function: extractMOV
 *Description: Given that we have a MOV header JUMP through the mov data structures
    until we reach EOF
*Return: A pointer to where the EOF of the MOV is in the current buffer
*****************************************************************************/
char* extractMOV(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec* needle,unsigned long long f_offset)
```

```
{
    char* buf=foundat-4;
    char* extractbuf = NULL;
    unsigned int atomsize=0;
    unsigned int filesize=0;
    int mdat=FALSE;
    foundat-=4;
    buflen+=4;
//    printf("mooooov\n");
    while(1) /*Loop through all the atoms until the EOF is reached*/
    {
        atomsize=htoi(foundat,FOREMOST_BIG_ENDIAN);
#ifdef DEBUG
        printf("Atomsize:=%d\n",atomsize);
#endif
        if(atomsize <= 0 || atomsize > needle->max_len)
          {
                    return foundat+needle->header_len+4;
          }
        filesize+=atomsize; /*Add the atomsize to the total file size*/
          //printf("mark2\n");
        if(filesize > buflen)
        {
          #ifdef DEBUG
                  printf("file size > buflen fs:=%d bf:=%lld\n",filesize, buflen);
          #endif
                    if(buflen >= needle->max_len) return foundat+needle->header_len+4;
                    else
                    {
                              //printf("buflen:=%lld max:=%lld",buflen,needle->max_len);
                              return NULL;
          }
          }
          //printf("mark4\n");
        foundat+=atomsize;
        if(buflen-(foundat-buf) < 5)
        {
            if(mdat)
            {
                break;
            }
            else
            {
#ifdef DEBUG
                printf("No mdat found");
#endif
                    return foundat;
            }
        }
          /*Check if we have an mdat atom, these are required thus can be used to
          * Weed out corrupted file*/
        if(strncmp(foundat+4,"mdat",4)==0)
        {
            mdat=TRUE;
        }

        if(checkMov(foundat+4)) /*Check to see if we are at a valid header*/
        {
#ifdef DEBUG
            printf("Checkmov succeeded\n");
#endif
        }
        else
        {
#ifdef DEBUG
            printf("Checkmov failed\n");
#endif
            if(mdat)
            {
                break;
            }
```

```
            else
            {
#ifdef DEBUG
                printf("No mdat found");
#endif
                return foundat;

            }
        }
    } //End loop
    if(foundat)
    {

        filesize = (foundat-buf);
#ifdef DEBUG
        printf("file size:=%d\n",filesize);
#endif
         extractbuf=(unsigned char*) malloc(filesize*sizeof(char));
         memcpy(extractbuf,buf,filesize);
         writeToDisk(s,needle,filesize,extractbuf,c_offset+f_offset);
        free(extractbuf);

         return foundat;
    }
#ifdef DEBUG
        printf("NULL Atomsize:=%d\n",atomsize);
#endif
    return NULL;

}



/*********************************************************************************
 *Function: extractWMV
 *Description: Given that we have a WMV header
    search for the file header and grab the file size.
*Return: A pointer to where the EOF of the WMV is in the current buffer
**********************************************************************************/

char* extractWMV(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec* needle,unsigned long long f_offset)
{

    char* currentpos=NULL;
    char* header=foundat;
    char* extractbuf = NULL;
    char* buf=foundat;
    unsigned long long int size=0;
    unsigned long long file_size=0;
    int headerSize=0;
    int fileObjHeaderSize=0;
    int numberofHeaderObjects=0;
    int reserved[2];
    int bytes_to_search=0;

    /*If we have less than a WMV header bail out*/
    if(buflen < 70) return NULL;

    foundat+=16;/*Jump to the header size*/
    headerSize=htoll(foundat,FOREMOST_LITTLE_ENDIAN);
    foundat+=8;
    numberofHeaderObjects=htoi(foundat,FOREMOST_LITTLE_ENDIAN);
    foundat+=4;                                  //Jump to the begin File properties obj
    reserved[0]=foundat[0];
    reserved[1]=foundat[1];
    foundat+=2;

//end header obj
//****************************************************/
```

```
                                                  //Sanity Check
    if(headerSize <= 0 || numberofHeaderObjects <= 0 || reserved[0] != 1)
    {
        return foundat;
    }

    currentpos=foundat;
    if(buflen-(foundat-buf) >= needle->max_len) bytes_to_search=needle->max_len;
    else bytes_to_search=buflen-(foundat-buf);

/*Note we are not searching for the footer here, just the file header ID so we can get
the file size*/
    foundat=  bm_search(needle->footer,needle->footer_len,foundat,bytes_to_search,needle-
>footer_bm_table,needle->case_sen,SEARCHTYPE_FORWARD);
    if(foundat)
    {
        foundat+=16;/*jump to the headersize*/
        fileObjHeaderSize=htoll(foundat,LITTLE_ENDIAN);
        foundat+=24;                              //Jump to the file size obj
        size=htoi(foundat,LITTLE_ENDIAN);
#ifdef DEBUG
         printf("SIZE:=%lld\n",size);
#endif
    }
    else
    {
        return NULL;
    }

/*Sanity check data*/
    if(size > 0 && size <= needle->max_len && size <= buflen)
    {
        header+=size;
#ifdef DEBUG
        printf("   Found a WMV at:=%lld,File size:=%lld\n",c_offset,size);
        printf("   Headersize:=%d, numberofHeaderObjects:= %d
,reserved:=%d,%d\n",headerSize,numberofHeaderObjects,reserved[0],reserved[1]);
#endif
         /*Everything seem ok, write to disk*/
        file_size = (header-buf);
        extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
         memcpy(extractbuf,buf,file_size);
         writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);
        free(extractbuf);
         foundat+=file_size;
        return header;
    }

    return NULL;

}

/*****************************************************************************
 *Function: extractRIFF
 *Description: Given that we have a RIFF header parse header and grab the file size.
 *Return: A pointer to where the EOF of the RIFF is in the current buffer
 *****************************************************************************/
char* extractRIFF(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec* needle,unsigned long long f_offset,char* type)
{
    unsigned char* buf=foundat;

    char* extractbuf =NULL;
    int size=0;
    unsigned long long file_size=0;

    size=htoi(&foundat[4],FOREMOST_LITTLE_ENDIAN);       /* Grab the total file size in
little endian from offset 4*/
    if(strncmp(&foundat[8],"AVI",3)==0)           /*Sanity Check*/
    {
        if(strncmp(&foundat[12],"LIST",4)==0)     /*Sanity Check*/
```

```
        {
            if(size > 0 && size <= needle->max_len && size <= buflen)
            {
#ifdef DEBUG
                printf("\n   Found an AVI at:=%lld,File size:=%d\n",c_offset,size);
#endif
                file_size = size;
                extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
                   memcpy(extractbuf,buf,file_size);
         needle->suffix="avi";
                   if(!strstr(needle->suffix,type) && type!="all") return foundat+size;
                   writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);
         free(extractbuf);
                foundat+=size;
                   return foundat;
            }
              return buf+needle->header_len;

        }
        else
        {
            return buf+needle->header_len;
        }
    }
    else if(strncmp(&foundat[8],"WAVE",4)==0)           /*Sanity Check*/
    {
        if(size > 0 && size <= needle->max_len && size <= buflen)
        {
#ifdef DEBUG
                printf("\n   Found a WAVE at:=%lld,File size:=%d\n",c_offset,size);
#endif

                file_size = size;
                   extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
                   memcpy(extractbuf,buf,file_size);
         needle->suffix="wav";
                   if(!strstr(needle->suffix,type) && type!="all") return foundat+size;

                   writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);
         free(extractbuf);
                   foundat+=file_size;
                  return foundat;
        }
         return buf+needle->header_len;

    }
    else
    {
        return buf+needle->header_len;
    }
    return NULL;

}

/********************************************************************************
 *Function: extractBMP
 *Description: Given that we have a BMP header parse header and grab the file size.
 *Return: A pointer to where the EOF of the BMP is in the current buffer
 ********************************************************************************/
char* extractBMP(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec* needle,unsigned long long f_offset)
{
    char* buf=foundat;
    int size=0;
    int headerlength=0;
    int verticalsize=0;
    char* extractbuf=NULL;
    unsigned long long file_size=0;

    foundat+=2;                                 /*JUMP the first to bytes of the header
(BM)*/
```

```
    size=htoi(foundat,LITTLE_ENDIAN);           /*Grab the total file size in
little_endian*/

                                                 /*Sanity Check*/
    if(size <= 0 || size > needle->max_len) return foundat;

    if(buflen-(foundat-buf) < 20)
    {
        return foundat;
    }
    foundat+=16;
    headerlength=htoi(foundat,FOREMOST_LITTLE_ENDIAN);

//Header length
    if(headerlength > 1000 || headerlength <= 0) return foundat;

    foundat+=4;
    verticalsize=htoi(foundat,FOREMOST_LITTLE_ENDIAN);

//Vertical length
    if(verticalsize <=0 || verticalsize > 2000) return foundat;

    foundat-=22;
#ifdef DEBUG
    printf("\n     The size of the BMP is %d, Header length:=%d , Vertical Size:=
%d\n",size,headerlength,verticalsize);
#endif
    if(size <= buflen)
    {
        file_size = size;
         extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
         memcpy(extractbuf,buf,file_size);
        writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);
        free(extractbuf);
         foundat+=file_size;
        return foundat;

    }
    return NULL;
}


/********************************************************************************
 *Function: extractGIF
 *Description: Given that we have a GIF header parse the given buffer to determine
 *        where the file ends.
 *Return: A pointer to where the EOF of the GIF is in the current buffer
 ********************************************************************************/

char* extractGIF(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec* needle,unsigned long long f_offset)
{
    char* buf=foundat;
    char* currentpos=foundat;
    char* extractbuf = NULL;
    int bytes_to_search=0;
    unsigned long long file_size=0;
//printf("needle->header_len:=%d needle->footer_len:=%d\n",needle->header_len,needle-
>footer_len);

    foundat+=4;                                  /*Jump the first 4 bytes of the gif
header (GIF8)*/


                                                 /*Check if the GIF is type 89a or 87a*/
    if(strncmp(foundat,"9a",2)==0 || strncmp(foundat,"7a",2)==0)
    {
        foundat+=2;                              /*Jump the length of the header*/

        currentpos=foundat;
         if(buflen-(foundat-buf) >= needle->max_len) bytes_to_search=needle->max_len;
```

```
          else bytes_to_search=buflen-(foundat-buf);
          //printf("bytes_to_search:=%d needle->footer_len:=%d needle-
>header_len:=%d\n",bytes_to_search,needle->footer_len,needle->header_len);
          foundat=  bm_search(needle->footer,needle-
>footer_len,foundat,bytes_to_search,needle->footer_bm_table,needle-
>case_sen,SEARCHTYPE_FORWARD);
        if(foundat)
        {
                    /*We found the EOF, write the file to disk and return*/
#ifdef DEBUG
          printx(foundat,0,16);
#endif
                    file_size = (foundat-buf)+needle->footer_len;
#ifdef DEBUG
                    printf("The GIF file size is  %llu
c_offset:=%llu\n",file_size,c_offset);
#endif
                    extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
                    memcpy(extractbuf,buf,file_size);
          writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);
          foundat+=needle->footer_len;
          free(extractbuf);
          return foundat;
        }
          return NULL;

    }
    else                                          /*Invalid GIF header return the current
pointer*/
    {
        return foundat;
    }

}

/*********************************************************************************
 *Function: extractMPG
 * Not done yet
 *********************************************************************************/

char* extractMPG(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec * needle,unsigned long long f_offset)
{
    char* buf=foundat;
    char* currentpos=NULL;

    unsigned char* extractbuf = NULL;
    //signed short headersize=0;
    int bytes_to_search=0;
    unsigned short size=0;
    unsigned long long file_size=0;
   /*
    size=htos(&foundat[4],FOREMOST_BIG_ENDIAN);
    printf("size:=%d\n",size);

    printx(foundat,0,16);
    foundat+=4;
    */

    int j=0;
    if(foundat[15]=='\xBB')
    {

    }
    else
    {

          return buf+needle->header_len;
    }

    if(buflen <=2*KILOBYTE)
```

```
	{
		bytes_to_search=buflen;
	}
	else
	{
		bytes_to_search=2*KILOBYTE;
	}
	while(1)
	{
		j=0;
		currentpos=foundat;
#ifdef DEBUG
			printf("Searching for marker\n");
#endif
		foundat=  bm_search(needle->markerlist[0].value,needle-
>markerlist[0].len,foundat,bytes_to_search,needle->markerlist[0].marker_bm_table,needle-
>case_sen,SEARCHTYPE_FORWARD);

		if(foundat)
		{
#ifdef DEBUG
			printf("Found after searching %d\n",foundat-currentpos);
#endif
			while(1)
			{

				if(foundat[3] >= '\xBB' && foundat[3] <='\xEF')
				{
#ifdef DEBUG
					printf("jumping %d:\n",j);
#endif
					size=htos(&foundat[4],FOREMOST_BIG_ENDIAN);
#ifdef DEBUG
					printf("\t hit: ");
					printx(foundat,0,16);
					printf("size:=%d\n\tjump: ",size);
#endif
					file_size+=(foundat-buf)+size;
					if(size <= 0 || size > buflen-(foundat-buf))
					{
#ifdef DEBUG
						printf("Not enough room in the buffer ");
#endif
						if(size <= 50*KILOBYTE && size > 0)
						{
							/*We should probably search
more*/
							if(file_size < needle->max_len)
							{
								return NULL;
							}
							else
							{
								break;
							}
						}
						else
						{
							return currentpos+needle-
>header_len;
						}
					}
					foundat+=size+6;
#ifdef DEBUG
					printx(foundat,0,16);
#endif
					j++;
				}
				else
				{
```

```
                                        break;
                                }
                        }
                        if(foundat[3]=='\xB9')
                        {
                                break;
                        }
                        else if(foundat[3]!='\xBA' && foundat[3]!='\x00')
                        {
                                /*This is the error state where this doesn't seem to be an
mpg anymore*/

                                size=htos(&foundat[4],FOREMOST_BIG_ENDIAN);
#ifdef DEBUG
                                printf("\t ***TEST: %x\n",foundat[3]);
                                printx(foundat,0,16);

                                printf("size:=%d\n",size);
#endif
                                if((currentpos - buf) >= 1*MEGABYTE)
                                {
                                        foundat=currentpos;
                                        break;
                                }
                                return currentpos+needle->header_len;

                        }
                        else if(foundat[3]=='\xB3')
                        {
                                //exit(-1);
                                foundat+=3;
                        }
                        else
                        {
                                foundat+=3;
                        }
                }
                else
                {
                        if((currentpos - buf) >= 1*MEGABYTE)
                        {
                                foundat=currentpos;
                                break;
                        }
                        else
                        {
#ifdef DEBUG
                                printf("RETURNING BUF\n");
#endif
                                return buf+needle->header_len;
                        }
                }
        }


        if(foundat)
        {
                file_size = (foundat-buf)+needle->footer_len;
                if(file_size < 1*KILOBYTE) return buf+needle->header_len;
        }
        else
        {
                return buf+needle->header_len;
               // file_size= needle->max_len;
        }
        if(file_size > buflen) file_size=buflen;
        foundat=buf;
        #ifdef DEBUG
                        printf("The file size is  %llu  c_offset:=%llu\n",file_size,c_offset);
        #endif
        extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
```

```
    memcpy(extractbuf,buf,file_size);
    writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);
    foundat+=file_size;
    free(extractbuf);
    return foundat;
}


/********************************************************************************
 *Function: extractJPEG
 *Description: Given that we have a JPEG header parse the given buffer to determine
 *        where the file ends.
 *Return: A pointer to where the EOF of the JPEG is in the current buffer
 ********************************************************************************/

char* extractJPEG(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec * needle,unsigned long long f_offset)
{
    char* buf=foundat;
    char* currentpos=NULL;

    unsigned char* extractbuf = NULL;
    signed short headersize=0;
    int bytes_to_search=0;
    int hasTable=FALSE;
    int hasHuffman=FALSE;
    unsigned long long file_size=0;


    /*Check if we have a valid header*/
    if(buflen < 128)
    {
          printf("low buffer %lld\n",buflen);
          return NULL;
    }
    if(foundat[3]=='\xe0');//JFIF header
    else if(foundat[3]=='\xe1');//EXIF header
    else return foundat+needle->header_len;//Invalid keep searching

    while(1) /* Jump through the headers until we reach the "data" part of the file*/
    {
#ifdef DEBUG
          printx(foundat,0,16);
#endif
        foundat+=2;
        headersize=htos(&foundat[2],FOREMOST_BIG_ENDIAN);
#ifdef DEBUG
          printf("Headersize:=%d buflen:=%lld\n",headersize,buflen);
#endif
        if(headersize < 0)
        {
#ifdef DEBUG
              printf("Negative header size\n");
#endif
            return buf+needle->header_len;;
        }

          if(headersize > buflen)
          {
                    return NULL;
          }
        foundat+=headersize;

        if(foundat[2]!='\xff')
        {
            break;
        }
         /*Ignore 2 "0xff" side by side*/
        if(foundat[2]=='\xff' && foundat[3]=='\xff')
        {
            foundat++;
```

```
        }
        if(foundat[3]=='\xdb' || foundat[4]=='\xdb')
        {
            hasTable=TRUE;
        }
        else if(foundat[3]=='\xc4')
        {
            hasHuffman=TRUE;
        }
    }

          /*All jpegs must contact a Huffman marker as well as a quantization table*/
    if(!hasTable || !hasHuffman)
    {
#ifdef DEBUG
printf("No Table or Huffman \n");
#endif
        return buf+needle->header_len;
    }

    currentpos=foundat;
    //sprintf("Searching for footer\n");
    if(buflen-(foundat-buf) >= needle->max_len) bytes_to_search=needle->max_len;
    else bytes_to_search=buflen-(foundat-buf);

    foundat=  bm_search(needle->footer,needle->footer_len,foundat,bytes_to_search,needle-
>footer_bm_table,needle->case_sen,SEARCHTYPE_FORWARD);


    if(foundat) /*Found found a valid JPEG*/
    {
          /*We found the EOF, write the file to disk and return*/
        file_size = (foundat-buf)+needle->footer_len;
#ifdef DEBUG
          printf("The jpeg file size is  %llu  c_offset:=%llu\n",file_size,c_offset);
#endif
          extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
          memcpy(extractbuf,buf,file_size);
        writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);
        foundat+=needle->footer_len;
          free(extractbuf);
          return foundat;
    }
    else
    {
          return NULL;
    }

} //End ExtractJPEG

/*****************************************************************************
 *Function: extractGENERIC
 *Description:
 *Return: A pointer to where the EOF of the
 *****************************************************************************/

char* extractGENERIC(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec * needle,unsigned long long f_offset)
{
    char* buf=foundat;
    unsigned char* extractbuf = NULL;
    int bytes_to_search=0;
    unsigned long long file_size=0;

    if(buflen-(foundat-buf) >= needle->max_len) bytes_to_search=needle->max_len;
    else bytes_to_search=buflen-(foundat-buf);

    if(needle->footer==NULL)
    {
          foundat=NULL;
    }
```

```
    else
    {
          foundat=  bm_search(needle->footer,needle-
>footer_len,foundat,bytes_to_search,needle->footer_bm_table,needle-
>case_sen,SEARCHTYPE_FORWARD);
    }

    if(foundat)
    {
              file_size = (foundat-buf)+needle->footer_len;
    }
    else
    {
              file_size= needle->max_len;
    }
    if(file_size > buflen) file_size=buflen;
    foundat=buf;
    #ifdef DEBUG
                   printf("The file size is  %llu  c_offset:=%llu\n",file_size,c_offset);
    #endif
    extractbuf=(unsigned char*) malloc(file_size*sizeof(char));
    memcpy(extractbuf,buf,file_size);
    writeToDisk(s,needle,file_size,extractbuf,c_offset+f_offset);
    foundat+=file_size;
    free(extractbuf);
    return foundat;
}

char* extractFile(f_state *s, unsigned long long c_offset,char *foundat,
unsigned long long buflen, s_spec * needle,unsigned long long f_offset)
{
          if(needle->type==JPEG)
          {
                   return extractJPEG(s,c_offset,foundat, buflen, needle,f_offset);
          }
          else if(needle->type==GIF)
          {
                   return extractGIF(s,c_offset,foundat, buflen, needle,f_offset);
          }
          else if(needle->type==BMP)
          {
                   return extractBMP(s,c_offset,foundat, buflen, needle,f_offset);
          }
          else if(needle->type==RIFF)
          {
                   needle->suffix="riff";
                   return extractRIFF(s,c_offset,foundat, buflen, needle,f_offset,"all");
          }
          else if(needle->type==AVI)
          {
                   return extractRIFF(s,c_offset,foundat, buflen, needle,f_offset,"avi");
          }
          else if(needle->type==WAV)
          {
                   return extractRIFF(s,c_offset,foundat, buflen, needle,f_offset,"wav");
                   needle->suffix="rif";
          }
          else if(needle->type==WMV)
          {
                   return extractWMV(s,c_offset,foundat, buflen, needle,f_offset);
          }
          else if(needle->type==OLE)
          {
                   needle->suffix="ole";
                   return extractOLE(s,c_offset,foundat, buflen, needle,f_offset,"all");
          }
          else if(needle->type==DOC)
          {
                   return extractOLE(s,c_offset,foundat, buflen, needle,f_offset,"doc");
          }
          else if(needle->type==PPT)
```

```
        {
                return extractOLE(s,c_offset,foundat, buflen, needle,f_offset,"ppt");
        }
        else if(needle->type==XLS)
        {
                return extractOLE(s,c_offset,foundat, buflen, needle,f_offset,"xls");
                needle->suffix="ole";
        }
        else if(needle->type==PDF)
        {
                return extractPDF(s,c_offset,foundat, buflen, needle,f_offset);
        }
        else if(needle->type==CPP)
        {
                return extractCPP(s,c_offset,foundat, buflen, needle,f_offset);
        }
        else if(needle->type==HTM)
        {
                return extractHTM(s,c_offset,foundat, buflen, needle,f_offset);
        }
        else if(needle->type==MPG)
        {
                return extractMPG(s,c_offset,foundat, buflen, needle,f_offset);
        }
        else if(needle->type==ZIP)
        {
                return extractZIP(s,c_offset,foundat, buflen, needle,f_offset);
        }
        else if(needle->type==MOV || needle->type==VJPEG)
        {
                return extractMOV(s,c_offset,foundat, buflen, needle,f_offset);
        }
        else if(needle->type==CONF)
        {
                return extractGENERIC(s,c_offset,foundat, buflen, needle,f_offset);
        }
        else
        {
                return NULL;
        }
}
```

B. EXTRACT.H

```
/*
        local file header signature     4 bytes  (0x04034b50)
         version needed to extract       2 bytes
         general purpose bit flag        2 bytes
         compression method              2 bytes
         last mod file time              2 bytes
         last mod file date              2 bytes
         crc-32                          4 bytes
         compressed size                 4 bytes
         uncompressed size               4 bytes
         filename length                 2 bytes
         extra field length              2 bytes
*/
/*
        central file header signature   4 bytes  (0x02014b50)
         version made by                 2 bytes
         version needed to extract       2 bytes
         general purpose bit flag        2 bytes
         compression method              2 bytes
         last mod file time              2 bytes
         last mod file date              2 bytes
         crc-32                          4 bytes
         compressed size                 4 bytes
         uncompressed size               4 bytes
         filename length                 2 bytes
         extra field length              2 bytes
         file comment length             2 bytes
```

```
        disk number start               2 bytes
        internal file attributes        2 bytes
        external file attributes        4 bytes
        relative offset of local header 4 bytes
*/
/* end of central dir signature    4 bytes  (0x06054b50)
        number of this disk             2 bytes
        number of the disk with the
        start of the central directory  2 bytes
        total number of entries in
        the central dir on this disk    2 bytes
        total number of entries in
        the central dir                 2 bytes
        size of the central directory   4 bytes
        offset of start of central
        directory with respect to
        the starting disk number        4 bytes
        zipfile comment length          2 bytes
        zipfile comment (variable size)
        */
struct zipLocalFileHeader {
    unsigned int signature; //0
    unsigned short version; //4
    unsigned short genFlag; //6
    signed short compression;//8
    unsigned short last_mod_time;//10
    unsigned short last_mod_date;//12
    unsigned int crc;//14
    unsigned int compressed;//18
    unsigned int uncompressed;//22
    unsigned short filename_length;//26
    unsigned short extra_length;//28
  };
  struct zipCentralFileHeader {
    unsigned int signature;//0
    unsigned char  version_extract[2];//4
    unsigned char  version_madeby[2];//6
    unsigned short genFlag;//8
    unsigned short compression;//10
    unsigned short last_mod_time;//12
    unsigned short last_mod_date;//14
    unsigned int crc;//16
    unsigned int compressed;//20
    unsigned int uncompressed;//24
    unsigned short filename_length;//28
    unsigned short extra_length;//30
    unsigned short filecomment_length;//32
    unsigned short disk_number_start;//34
  };
   struct zipEndCentralFileHeader {
    unsigned int signature;//0
    unsigned short numOfdisk;//4
    unsigned short compression;//6
    unsigned short start_of_central_dir;//8
    unsigned short num_entries_in_central_dir;//10
    unsigned int size_of_central_dir;//12
    unsigned int offset;//16
    unsigned short comment_length;//20
  };

void printZip(struct zipLocalFileHeader* fileHeader , struct zipCentralFileHeader*
centralHeader)
{
        printf("\n     Local Header Data\n");
        printf("GenFlag:=%d,compressed:=%d,uncompressed:=%d\n",fileHeader-
>genFlag,fileHeader->compressed,fileHeader->uncompressed);
        printf("Compression:=%d, filename_len:=%d,extralen:=%d\n",fileHeader-
>compression,fileHeader->filename_length,fileHeader->extra_length);

        printf("       Central Header Data\n");
```

```
        printf("GenFlag:=%d,compressed:=%d,uncompressed:=%d\n",centralHeader-
>genFlag,centralHeader->compressed,centralHeader->uncompressed);
        printf("Compression:=%d, Version Madeby:=%x%x\n",centralHeader-
>compression,centralHeader->version_madeby[0],centralHeader->version_madeby[1]);
}
```

??????spacing???

C. API.C

```
/*
        Modified API from http://chicago.sourceforge.net/devel/docs/ole/
        Basically the same API, added error checking and the ability
        to check buffers for docs except just files.
*/
#include "main.h"
#include "ole.h"

char buffer[OUR_BLK_SIZE];
char *extract_name;
int extract = 0;
int dir_count = 0;
int *FAT;
int verbose = TRUE;
int FATblk;
int currFATblk;
int highblk=0;
int block_list[OUR_BLK_SIZE / sizeof (int)];
extern int errno;



void initOLE()
{
    int i=0;
    extract=0;
    dir_count=0;
    FAT=NULL;
    highblk=0;
    FATblk=0;
    currFATblk=-1;
    dirlist=NULL;
    dl=NULL;
    for(i=0;i<OUR_BLK_SIZE / sizeof (int);i++)
    {
        block_list[i]=0;
    }
    for(i=0;i<OUR_BLK_SIZE;i++)
    {
        buffer[i]=0;
    }
}


void *
Malloc (size_t bytes)
{
    void *x;

    x = malloc (bytes);
    if (x)
        return x;
    die ("Can't malloc %d bytes.\n", (char *) bytes);
    return 0;
}
```

```
int
Read (int fd, char *buf, int size)
{
    if (read (fd, buf, size) != size)
    {
        fprintf (stderr, "Bad read of %d bytes\n", size);
        exit (1);
    }
    return size;
}


int
Write (int fd, char *buf, int size)
{
    if (write (fd, buf, size) != size)
    {
        fprintf (stderr, "Bad write of %d bytes\n", size);
        exit (1);
    }
    return size;
}


void
die (char *fmt, void *arg)
{
    fprintf (stderr, fmt, arg);
    exit (1);
}


int
get_dir_block (char* fd, int blknum,int buffersize)
{
    int i;
    struct OLE_DIR *dir;
    char* dest=NULL;

    dest=get_ole_block (fd, blknum,buffersize);
    if(dest==NULL)
    {
        return FALSE;
    }
    for (i = 0; i < DIRS_PER_BLK; i++)
    {
        dir = (struct OLE_DIR *) &dest[sizeof (struct OLE_DIR) * i];
        if (dir->type == NO_ENTRY)
            break;
    }
    if(i==DIRS_PER_BLK)
    {
        return TRUE;
    }
    else
    {
        return SHORT_BLOCK;
    }
}


int
get_dir_info (char *src)
{
    int i, j;
    char *p, *q;
    struct OLE_DIR *dir;
    int punctCount=0;
    short name_size=0;
```

```
    for (i = 0; i < DIRS_PER_BLK; i++)
    {
        dir = (struct OLE_DIR *) &src[sizeof (struct OLE_DIR) * i];
        punctCount=0;

        //if(dir->reserved!=0) return FALSE;
        if(dir->type < 0 )                          //Should we check if values are > 5
?????
        {
#ifdef DEBUG
            printf("\n Invalid directory type\n");
            printf("type:=%c size:=%lu \n", dir->type,dir->size);
#endif
            return FALSE;
        }

        if (dir->type == NO_ENTRY)
            break;

#ifdef DEBUG
//dump_dirent (i);
#endif
        dl = &dirlist[dir_count++];
        if(dl==NULL)
        {
#ifdef DEBUG
        printf("dl==NULL!!! bailing out\n");
#endif
                return FALSE;
        }

        if(dir_count > 500) return FALSE;          /*SANITY CHECKING*/
        q = dl->name;
        p = dir->name;

        name_size=htos((char*) &dir->namsiz,FOREMOST_LITTLE_ENDIAN);

#ifdef DEBUG
        printf(" dir->namsiz:=%d\n",name_size);
#endif
        if(name_size > 64|| name_size <= 0) return FALSE;

        if (*p < ' ')
            p += 2;                                /* skip leading short */
        for (j = 0; j < name_size; j++, p++)
        {

            if(p==NULL || q==NULL) return FALSE;
            if (*p && isprint(*p))
            {

                if(ispunct(*p)) punctCount++;
                *q++ = *p;

            }
        }

        if(punctCount > 3)
        {
#ifdef DEBUG
printf("dl->name:=%s\n",dl->name);
        printf("pcount > 3!!! bailing out\n");
#endif
            return FALSE;
        }

        if(dl->name==NULL)
        {
#ifdef DEBUG
            printf("  ***NULL dir name. bailing out \n");
```

```
#endif
            return FALSE;
        }



        /*Ignore Catalogs*/
        if(strstr(dl->name,"Catalog")) return FALSE;
        *q = 0;
        dl->type = dir->type;
        dl->size = htoi((char*)&dir->size,FOREMOST_LITTLE_ENDIAN);

        dl->start_block = htoi((char*)&dir->start_block,FOREMOST_LITTLE_ENDIAN);
        dl->next = htoi((char*)&dir->next_dirent,FOREMOST_LITTLE_ENDIAN);
        dl->prev = htoi((char*)&dir->prev_dirent,FOREMOST_LITTLE_ENDIAN);
        dl->dir = htoi((char*)&dir->dir_dirent,FOREMOST_LITTLE_ENDIAN);
        if (dir->type != STREAM)
        {
            dl->s1 = dir->secs1;
            dl->s2 = dir->secs2;
            dl->d1 = dir->days1;
            dl->d2 = dir->days2;
        }
    }
    return TRUE;
}




static int *lnlv;                                 /* last next link visited ! */
int
reorder_dirlist (struct DIRECTORY *dir, int level)
{
//printf("     Reordering the dirlist\n");
    dir->level = level;
    if (dir->dir != -1 || dir->dir > dir_count)
    {
        return 0;
    }
    else if (!reorder_dirlist (&dirlist[dir->dir], level + 1))
        return 0;
/* reorder next-link subtree, saving the most next link visited */
    if (dir->next != -1)
    {
        if (dir->next > dir_count)
            return 0;
        else if (!reorder_dirlist (&dirlist[dir->next], level))
            return 0;
    }
    else
        lnlv = &dir->next;
/* move the prev child to the next link and reorder it, if any exist
 */
    if (dir->prev != -1)
    {
        if (dir->prev > dir_count)
            return 0;
        else
        {
            *lnlv = dir->prev;
            dir->prev = -1;
            if (!reorder_dirlist (&dirlist[*lnlv], level))
                return 0;
        }
    }
    return 1;
}


int get_block (char* fd, int blknum, char *dest,long long int buffersize)
```

```
{
    char* temp=fd;
    int i=0;
    unsigned long long  jump=(unsigned long long) OUR_BLK_SIZE*(unsigned long
long)(blknum + 1);
    if(blknum < -1 || jump < 0 || blknum > buffersize || buffersize < jump)
    {
#ifdef DEBUG
        printf("       Bad blk read1 blknum:=%d  jump:=%lld
buffersize=%lld\n",blknum,jump,buffersize);
#endif
        return FALSE;
    }
    temp=fd+jump;
#ifdef DEBUG
printf("       Jumping to %lld blknum=%d buffersize=%lld\n",jump,blknum,buffersize);
#endif
    for(i=0;i < OUR_BLK_SIZE;i++)
    {
        dest[i]=temp[i];
    }
    if((blknum+1) > highblk) highblk=blknum+1;
    return TRUE;
}

char* get_ole_block (char* fd, int blknum,unsigned long long buffersize)
{
    unsigned long long  jump=(unsigned long long) OUR_BLK_SIZE*(unsigned long
long)(blknum + 1);
    if(blknum < -1 || jump < 0 || blknum > buffersize || buffersize < jump)
    {
#ifdef DEBUG
        printf("       Bad blk read1 blknum:=%d  jump:=%lld
buffersize=%lld\n",blknum,jump,buffersize);
#endif
        return FALSE;
    }
#ifdef DEBUG
printf("       Jumping to %lld blknum=%d buffersize=%lld\n",jump,blknum,buffersize);
 #endif
    return (fd+jump);
}



int
get_FAT_block (char* fd, int blknum, int *dest,int buffersize)
{
    static int FATblk;
 //   static int currFATblk = -1;

    FATblk = htoi((char*) &FAT[blknum / (OUR_BLK_SIZE / sizeof
(int))],FOREMOST_LITTLE_ENDIAN);
#ifdef DEBUG
       printf("****blknum:=%d FATblk:=%d currFATblk:=%d\n",blknum,FATblk,currFATblk);

#endif
    if (currFATblk != FATblk)
    {
#ifdef DEBUG
       printf("*****blknum:=%d FATblk:=%d\n",blknum,FATblk);

#endif
        if(!get_block (fd, FATblk, (char *) dest,buffersize))
        {
            return FALSE;
        }
        currFATblk = FATblk;
    }
    return TRUE;
}
```

```
void
dump_header (struct OLE_HDR *h )
{
    int i, *x;
    //struct OLE_HDR *h = (struct OLE_HDR *) buffer;

    // fprintf (stderr, "clsid  = ");
    //printx(h->clsid,0,16);
    fprintf (stderr, "\nuMinorVersion  = %u\t", htos((char*)&h-
>uMinorVersion,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "uDllVersion  = %u\t", htos((char*) &h-
>uDllVersion,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "uByteOrder  = %u\n", htos((char*) &h-
>uByteOrder,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "uSectorShift  = %u\t", htos((char *) &h-
>uSectorShift,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "uMiniSectorShift  = %u\t", htos((char *) &h-
>uMiniSectorShift,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "reserved  = %u\n", htos((char *) &h-
>reserved,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "reserved1  = %u\t", htoi((char *) &h-
>reserved1,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "reserved2  = %u\t", htoi((char *) &h-
>reserved2,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "csectMiniFat = %u\t",htoi((char *) &h-
>csectMiniFat,FOREMOST_LITTLE_ENDIAN));
     fprintf (stderr, "miniSectorCutoff = %u\n",htoi((char *) &h-
>miniSectorCutoff,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "root_start_block  = %u\n", htoi((char *) &h-
>root_start_block,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "dir flag = %u\n", htoi((char *) &h-
>dir_flag,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "# FAT blocks = %u\n", htoi((char *) &h-
>num_FAT_blocks,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "FAT_next_block = %u\n", htoi((char *) &h-
>FAT_next_block,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "# extra FAT blocks = %u\n", htoi((char *) &h-
>num_extra_FAT_blocks,FOREMOST_LITTLE_ENDIAN));
    x = (int *) &h[1];
    fprintf (stderr, "bbd list:");
    for (i = 0; i < 109; i++, x++)
    {
        if ((i % 10) == 0)
            fprintf (stderr, "\n");
        if(*x=='\xff') break;
        fprintf (stderr, "%x ", *x);
    }
    fprintf (stderr, "\n      **************End of header***********\n");
}

struct OLE_HDR* reverseBlock(struct OLE_HDR *dest,struct OLE_HDR *h)
{
  int i, *x,*y;
  dest->uMinorVersion=htos((char*)&h->uMinorVersion,FOREMOST_LITTLE_ENDIAN);
  dest->uDllVersion=htos((char*) &h->uDllVersion,FOREMOST_LITTLE_ENDIAN);
  dest->uByteOrder=htos((char*) &h->uByteOrder,FOREMOST_LITTLE_ENDIAN);    /*28*/
  dest->uSectorShift=htos((char *) &h->uSectorShift,FOREMOST_LITTLE_ENDIAN);
  dest->uMiniSectorShift=htos((char *) &h-
>uMiniSectorShift,FOREMOST_LITTLE_ENDIAN);/*32*/
  dest->reserved=htos((char *) &h->reserved,FOREMOST_LITTLE_ENDIAN);       /*34*/
  dest->reserved1=htoi((char *) &h->reserved1,FOREMOST_LITTLE_ENDIAN);     /*36*/
  dest->reserved2=htoi((char *) &h->reserved2,FOREMOST_LITTLE_ENDIAN);     /*40*/

  dest->num_FAT_blocks=htoi((char *) &h->num_FAT_blocks,FOREMOST_LITTLE_ENDIAN);
        /*44*/
  dest->root_start_block=htoi((char *) &h->root_start_block,FOREMOST_LITTLE_ENDIAN);
/*48*/
  dest->dfsignature=htoi((char *) &h->dfsignature,FOREMOST_LITTLE_ENDIAN);
        /*52*/
```

```
  dest->miniSectorCutoff=htoi((char *) &h->miniSectorCutoff,FOREMOST_LITTLE_ENDIAN);
             /*56*/
  dest->dir_flag=htoi((char *) &h->dir_flag,FOREMOST_LITTLE_ENDIAN);
       /*60 first sec in the mini fat chain*/
  dest->csectMiniFat=htoi((char *) &h->csectMiniFat,FOREMOST_LITTLE_ENDIAN);       /*64
number of sectors in the minifat */
  dest->FAT_next_block=htoi((char *) &h->FAT_next_block,FOREMOST_LITTLE_ENDIAN);   /*68*/
  dest->num_extra_FAT_blocks=htoi((char *) &h-
>num_extra_FAT_blocks,FOREMOST_LITTLE_ENDIAN);

  x = (int *) &h[1];
  y= (int *) &dest[1];
  for (i = 0; i < 109; i++, x++)
  {
        *y=htoi((char *) x,FOREMOST_LITTLE_ENDIAN);
       y++;
  }
  return dest;
}

void dump_ole_header (struct OLE_HDR *h )
{
    int i, *x;


    //fprintf (stderr, "clsid  = ");
    //printx(h->clsid,0,16);
    fprintf (stderr, "\nuMinorVersion  = %u\t", htos((char*)&h-
>uMinorVersion,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "uDllVersion  = %u\t", htos((char*) &h-
>uDllVersion,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "uByteOrder  = %u\n",  htos((char*) &h-
>uByteOrder,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "uSectorShift  = %u\t", htos((char *) &h-
>uSectorShift,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "uMiniSectorShift  = %u\t", htos((char *) &h-
>uMiniSectorShift,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "reserved  = %u\n", htos((char *) &h-
>reserved,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "reserved1  = %u\t", htoi((char *) &h-
>reserved1,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "reserved2  = %u\t", htoi((char *) &h-
>reserved2,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "csectMiniFat = %u\t",htoi((char *) &h-
>csectMiniFat,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "miniSectorCutoff = %u\n",htoi((char *) &h-
>miniSectorCutoff,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "root_start_block  = %u\n", htoi((char *) &h-
>root_start_block,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "dir flag = %u\n", htoi((char *) &h-
>dir_flag,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "# FAT blocks = %u\n", htoi((char *) &h-
>num_FAT_blocks,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "FAT_next_block = %u\n", htoi((char *) &h-
>FAT_next_block,FOREMOST_LITTLE_ENDIAN));
    fprintf (stderr, "# extra FAT blocks = %u\n", htoi((char *) &h-
>num_extra_FAT_blocks,FOREMOST_LITTLE_ENDIAN));
    x = (int *) &h[1];
    fprintf (stderr, "bbd list:");
    for (i = 0; i < 109; i++, x++)
    {
        if ((i % 10) == 0)
            fprintf (stderr, "\n");
        if(*x=='\xff') break;
        fprintf (stderr, "%x ", htoi((char *) x,FOREMOST_LITTLE_ENDIAN));
    }
    fprintf (stderr, "\n      **************End of header***********\n");
}


int
```

```
dump_dirent (int which_one)
{
    int i;
    char *p;
    short unknown;
    struct OLE_DIR *dir;

    dir = (struct OLE_DIR *) &buffer[which_one * sizeof (struct OLE_DIR)];
    if (dir->type == NO_ENTRY)
        return TRUE;
    fprintf (stderr, "DIRENT_%d :\t", dir_count);
    fprintf (stderr, "%s\t", (dir->type == ROOT) ? "root directory" :
    (dir->type == STORAGE) ? "directory" : "file");

/* get UNICODE name */
    p = dir->name;
    if (*p < ' ')
    {
        unknown = *((short *) p);
        //fprintf (stderr, "%04x\t", unknown);
        p += 2;                                     /* step over unknown short */
    }
    for (i = 0; i < dir->namsiz; i++, p++)
    {
        if (*p && (*p > 0x1f))
        {
            if(isprint(*p))
            {
                fprintf (stderr, "%c", *p);
            }
            else
            {
                printf("***   Invalid char %x ***\n",*p);
                return FALSE;
            }
        }
    }
    fprintf (stderr, "\n");
    //fprintf (stderr, "prev_dirent = %lu\t", dir->prev_dirent);
    //fprintf (stderr, "next_dirent = %lu\t", dir->next_dirent);
    //fprintf (stderr, "dir_dirent  = %lu\n", dir->dir_dirent);
    //fprintf (stderr, "name  = %s\t", dir->name);
    fprintf (stderr, "namsiz  = %u\t", dir->namsiz);
    fprintf (stderr, "type  = %d\t", dir->type);
    fprintf (stderr, "reserved  = %u\n", dir->reserved);

    fprintf (stderr, "start block  = %lu\n", dir->start_block);
    fprintf (stderr, "size  = %lu\n", dir->size);
    fprintf (stderr, "\n      **************End of dirent***********\n");
    return TRUE;
}
```

D. OLE.H

```
#define TRUE                  1
#define FALSE                 0
#define SPECIAL_BLOCK           -3
#define END_OF_CHAIN            -2
#define UNUSED                  -1

#define NO_ENTRY                0
#define STORAGE                 1
#define STREAM                  2
#define ROOT                    5
#define SHORT_BLOCK           3

#define FAT_START               0x4c
#define OUR_BLK_SIZE            512
#define DIRS_PER_BLK            4
#define MIN(x,y)                ((x) < (y) ? (x) : (y))
```

```
#include <stdarg.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <ctype.h>

struct OLE_HDR
{
  char magic[8];       /*0*/
  char clsid[16];      /*8*/
  unsigned short uMinorVersion;       /*24*/
  unsigned short uDllVersion; /*26*/
  unsigned short uByteOrder;  /*28*/
  unsigned short uSectorShift;        /*30*/
  unsigned short uMiniSectorShift;/*32*/
  unsigned short reserved;    /*34*/
  unsigned long  reserved1;   /*36*/
  unsigned long  reserved2;   /*40*/
  unsigned long  num_FAT_blocks;              /*44*/
  unsigned long  root_start_block;      /*48*/
  unsigned long  dfsignature;                 /*52*/
  unsigned long  miniSectorCutoff;                    /*56*/
  unsigned long  dir_flag;                    /*60 first sec in the mini fat chain*/
  unsigned long csectMiniFat; /*64 number of sectors in the minifat */
  unsigned long FAT_next_block;       /*68*/
  unsigned long num_extra_FAT_blocks;   /*72*/

  /* FAT block list starts here !! first 109 entries  */
};


struct OLE_DIR
{
  char name[64];
  unsigned short namsiz;
  char type;
  char bflags;  //0 or 1
  unsigned long prev_dirent;
  unsigned long next_dirent;
  unsigned long dir_dirent;
  char clsid[16];
  unsigned long userFlags;
  int secs1;
  int days1;
  int secs2;
  int days2;
  unsigned long start_block; //starting SECT of stream
  unsigned long size;
  short reserved; //must be 0
};

struct DIRECTORY
{
  char name[64];
  int type;
  int level;
  int start_block;
  int size;
  int next;
  int prev;
  int dir;
  int s1;
  int s2;
  int d1;
  int d2;
}
 *dirlist, *dl;
```

```
int get_dir_block (char* fd, int blknum,int buffersize);
int get_dir_info (char *src);
void extract_stream (char* fd, int blknum, int size);
void dump_header (struct OLE_HDR *h );
int dump_dirent (int which_one);
int get_block (char* fd, int blknum, char *dest,long long int buffersize);
int get_FAT_block (char* fd, int blknum,int* dest,int buffersize);
int reorder_dirlist (struct DIRECTORY *dir, int level);

char* get_ole_block (char* fd, int blknum,unsigned long long buffersize);
struct OLE_HDR* reverseBlock(struct OLE_HDR *dest,struct OLE_HDR *h);

void dump_ole_header (struct OLE_HDR *h );
void *Malloc (size_t bytes);
int Read (int fd, char *buf, int size);
int Write (int fd, char *buf, int size);
void die (char *fmt, void *arg);
void initOLE();
```

E. ENGINE.C

```
/* FOREMOST
 *
 * By Jesse Kornblum and Kris Kendall
 *
 * This is a work of the US Government. In accordance with 17 USC 105,
 * copyright protection is not available for any work of the US Government.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *
 */

#include "main.h"



int user_interrupt(f_state *s, f_info *i)
{
  audit_msg(s,"Interrupt received at %s", current_time());

  /* RBF - Write user_interrupt */
  fclose(i->handle);
  free(s);
  free(i);
  exit(-1);
  return FALSE;
}


char* grabFromDisk(unsigned long long  offset, f_info *i,unsigned long long length)
{

    unsigned long long bytesread = 0;
    char* newbuf = (char*) malloc(length*sizeof(char));
```

```
    fseeko(i->handle,offset,SEEK_SET);
    bytesread = fread(newbuf,1,length,i->handle);
    if(bytesread!=length) return NULL;
    else     return newbuf;
}

/*
   Perform a modified boyer-moore string search (w/ support for wildcards and case-
insensitive searches)
   and allows the starting position in the buffer to be manually set, which allows data
to be skipped
*/

unsigned char *bm_search_skipn(char *needle, size_t needle_len,unsigned char *haystack,
size_t haystack_len,
        size_t table[UCHAR_MAX + 1], int casesensitive,int searchtype, int start_pos)
{
    register size_t shift=0;
    register size_t pos = start_pos;
    unsigned char *here;

    if(needle_len == 0)
        return haystack;


    if (searchtype == SEARCHTYPE_FORWARD || searchtype == SEARCHTYPE_FORWARD_NEXT )
    {
        while (pos < haystack_len)
        {
            while( pos < haystack_len && (shift = table[(unsigned char)haystack[pos]]) >
0)
            {
                pos += shift;
            }
            if (0 == shift)
            {
                here = (char *)&haystack[pos-needle_len+1];

                if (0 == memwildcardcmp(needle,here, needle_len, casesensitive))
                {
                   return(here);
                }
                else  pos++;
            }
        }
        return NULL;
    }
    else if(searchtype == SEARCHTYPE_REVERSE)          //Run our search backwards
    {
        while (pos < haystack_len)
        {
            while(    pos    <    haystack_len    &&    (shift    =    table[(unsigned
char)haystack[haystack_len-pos-1]]) > 0)
            {
                pos += shift;
            }
            if (0 == shift)
            {
                if (0 == memwildcardcmp(needle,here = (char *)&haystack[haystack_len-pos-
1], needle_len, casesensitive))
                {
                    return(here);
                }
                else  pos++;
            }
        }
        return NULL;
    }
    return NULL;
}
```

```
/*
   Perform a modified boyer-moore string search (w/ support for wildcards and case-
insensitive searches)
   and allows the starting position in the buffer to be manually set, which allows data
to be skipped
*/

unsigned char *bm_search(char *needle, size_t needle_len,unsigned char *haystack, size_t
haystack_len,
size_t table[UCHAR_MAX + 1], int case_sen,int searchtype)
{
   //printf("The needle2 is:\t");
   //printx(needle,0,needle_len);

    return bm_search_skipn(needle,
        needle_len,
        haystack,
        haystack_len,
        table,
        case_sen,
        searchtype,
        needle_len - 1);

}

void setup_stream(f_state *s, f_info *i)
{
  char buffer[MAX_STRING_LENGTH];
  unsigned long long skip=(((unsigned long long)s->skip)*((unsigned long long)s-
>block_size));
#ifdef DEBUG
  printf("s->skip=%d   s->block_size=%d   total=%llu\n",s->skip,s->block_size,(((unsigned
long long)s->skip)*((unsigned long long)s->block_size)));
#endif
  i->bytes_read = 0;
  i->total_megs = i->total_bytes / ONE_MEGABYTE;

  if (i->total_bytes != 0)
    audit_msg(s,"Length: %s (%llu bytes)",
             human_readable(i->total_bytes,buffer),i->total_bytes);
  else
    audit_msg(s,"Length: Unknown");

if(s->skip!=0)
{
         audit_msg(s,"Skipping: %s (%llu bytes)",
             human_readable(skip,buffer),skip);
         fseeko(i->handle,skip,SEEK_SET);
         if(i->total_bytes!=0) i->total_bytes-=skip;
}
  audit_msg(s," ");

#ifdef __WIN32
  i->last_read = 0;
  i->overflow_count = 0;
#endif

}


int indBlock(char* foundat,unsigned long long buflen,int bs)
{

       unsigned char* temp=foundat;
       int jump=12*bs;
       unsigned  int block=0;
       unsigned  int block2=0;
       unsigned int dif=0;
       int i=0;
```

```
        unsigned int one=1;
        //int reconstruct=FALSE;

        /*Make sure we don't jump past the end of the buffer*/
        if(buflen < jump+16) return FALSE;

        while(i < bs/4)
        {
                block=htoi(&temp[jump+(i*4)],FOREMOST_LITTLE_ENDIAN);

                if(block < 0) return FALSE;
                if(block==0)
                {
                        break;
                }
                i++;
                block2=htoi(&temp[jump+(i*4)],FOREMOST_LITTLE_ENDIAN);
                if(block2 < 0)          return FALSE;

                if(block2==0)
                {
                        break;
                }

                dif=block2-block;

                if(dif==one)
                {
                        //printf("DIF==1\n");
#ifdef DEBUG
                        printf("block1:=%u, block2:=%u dif=%u\n",block,block2,dif);
#endif
                }
                else
                {
                        return FALSE;
                }
#ifdef DEBUG
                printf("block1:=%u, block2:=%u dif=%u\n",block,block2,dif);
#endif
        }
        if(i==0) return FALSE;

        /*Check if the rest of the bytes are zero'd out */
        for(i=i+1;i < bs/4;i++)
        {
                block=htoi(&temp[jump+(i*4)],FOREMOST_LITTLE_ENDIAN);
                if(block!=0)
                {
                        return FALSE;
                }
        }

        return TRUE;
}
//*********************************************************************************
***********************************************/

int  search_chunk(f_state*  s,  unsigned  char*  buf,   f_info  *i,  unsigned  long  long
chunk_size, unsigned long long f_offset)
{

    unsigned long long c_offset = 0;
    unsigned char* foundat=buf;
    unsigned char* current_pos=NULL;
    unsigned char* header_pos=NULL;
    unsigned char* newbuf=NULL;

    unsigned long long current_buflen=chunk_size;
    int tryBS[3]={4096,1024,512};
    s_spec * needle=NULL;
```

```
    int j=0;
    int bs=0;
    int rem=0;
    int x=0;
    for(j=0;j< s->num_builtin;j++)
    {
        needle=&search_spec[j];
        foundat=buf; /*reset the buffer for the next search spec*/
#ifdef DEBUG
        printf("       SEARCHING FOR %s's\n",needle->suffix);
#endif
        bs=0;
        current_buflen=chunk_size;
        while(foundat)
        {
                current_buflen=chunk_size-(foundat-buf);
#ifdef DEBUG
                    printf("current buf:=%lld\n",current_buflen);
#endif
                if (signal_caught == SIGTERM || signal_caught == SIGINT)
                {
                        user_interrupt(s,i);
                        printf ("Cleaning up.\n");
                        signal_caught = 0;
                }
                if(get_mode(s,mode_quiet))/*RUN QUIET SEARCH*/
                {
#ifdef DEBUG
                        printf("quick mode is on\n");
#endif
                        /*Check if we are not on a block head, adjust if so*/
                        rem=(foundat-buf) % s->block_size;
                        if(rem !=0)
                        {
                                foundat+=(s->block_size-rem);
                        }

                        if(memwildcardcmp(needle->header,foundat,needle->header_len,needle-
>case_sen)!=0)
                        {
                                /*No match, jump to the next block*/
                                //printf("     No match jumping bs\n");
                                if(current_buflen > s->block_size)
                                {
                                        foundat+=s->block_size;
                                        continue;
                                }
                                else  /*We  are  out  of  buffer  lets  go  to  the  next  search
spec*/
                                {
                                        foundat=NULL;
                                        break;
                                }
                        }
                }
                else /*RUN STANDARD SEARCH*/
                {

                        //printf("current buf:=%lld\n",current_buflen);
                        foundat = bm_search(needle->header,
                                needle->header_len,
                                foundat,
                                current_buflen,                       //How  much  to  search
through
                                needle->header_bm_table,
                                needle->case_sen,                //casesensative
                                SEARCHTYPE_FORWARD);

                        header_pos=foundat;
                }
```

```
		if(foundat != NULL && foundat >= 0)		/*We got something, run the 
appropriate heuristic to find the EOF*/
		{
		    current_buflen=chunk_size-(foundat-buf);
#ifdef DEBUG
//		    printf("current buf2:=%lld\n",current_buflen);
#endif

		    if(get_mode(s,mode_ind_blk))
		    {
#ifdef DEBUG
		//	printf("ind blk detection on\n");
#endif

			for(x=0;x<3;x++)
			{
				bs=tryBS[x];

				if(indBlock(foundat,current_buflen,bs))
				{
					if(get_mode(s,mode_verbose))
					 {
						audit_msg(s,"\n	Indirect Block Found 
using bs:=%d in a %s\n",bs,needle->suffix);


					}
#ifdef DEBUG
					printf("performing mem move\n");
#endif
					if(!memmove(foundat + 12*bs, foundat + 13*bs, 
current_buflen - 13*bs)) break;


#ifdef DEBUG
					printf("performing mem move complete\n");
#endif
					current_buflen-=bs;
					//current_buflen=chunk_size-(foundat-buf);
					break;
				}

			}

		    }



		     c_offset = (foundat-buf);
		    current_pos=foundat;
		    foundat=extractFile(s,c_offset,foundat,		current_buflen, 
needle,f_offset);

		    if(!foundat)
		    {
			if(current_buflen < needle->max_len)/*We need to bridge the gap*/
			{
#ifdef DEBUG
				printf("	Bridge the gap\n");
#endif
				//grow buffer, call again
				newbuf=grabFromDisk(c_offset+f_offset,i,needle->max_len);
				if(newbuf==NULL) break;
				current_pos=extractFile(s,c_offset,current_pos, 
current_buflen, needle,f_offset);
				if(!current_pos)
				{
					/*We failed so we should put the file* back*/
					fseeko(i->handle,c_offset+f_offset,SEEK_SET);
				}
				free(newbuf);
```

```
                      }
                      else
                      {
#ifdef DEBUG
                              printf("       RESET the FILE*\n");
#endif
                              foundat=header_pos;/*reset  the  foundat  pointer  to  the
location of the last header*/
                              foundat+=needle->header_len+1;/*jump past the header*/
                      }
                  }

              }
        }//end while
    }
    return TRUE;
}

int search_stream(f_state *s, f_info *i)
{
  unsigned long long bytesread =0;
  unsigned long long f_offset=0;
  unsigned long long chunk_size=((unsigned long long) s->chunk_size)*MEGABYTE;
  unsigned char* buf=(unsigned char *)malloc(sizeof(char)*chunk_size);

  setup_stream(s,i);
#ifdef DEBUG
        printf("\n\t READING THE FILE INTO MEMORY\n");
#endif
  while((bytesread = fread(buf,1,chunk_size,i->handle)) > 0)
  {
        if (signal_caught == SIGTERM || signal_caught == SIGINT)
         {
                user_interrupt(s,i);
                printf ("Cleaning up.\n");
                signal_caught = 0;
        }
#ifdef DEBUG
        printf("\n\tbytes_read:=%llu\n",bytesread);
#endif
        search_chunk(s,buf,i,bytesread,f_offset);
        f_offset+=bytesread;
        displayPosition(s,i,f_offset);
        /*
        f_offset-=50;//jump back 50 bytes to make sure we don't miss anything
        fseeko(i->handle,f_offset,SEEK_SET);
        */
  }
#ifdef DEBUG
        printf("\n\tDONE READING bytes_read:=%llu\n",bytesread);
#endif
  if (signal_caught == SIGTERM || signal_caught == SIGINT)
  {
    user_interrupt(s,i);
    printf ("Cleaning up.\n");
    signal_caught = 0;
  }
  free(buf);
  return FALSE;
}

void audit_start(f_state *s, f_info *i)
{
  audit_msg(s,FOREMOST_DIVIDER);
  audit_msg(s,"File: %s", i->file_name);
  audit_msg(s,"Start: %s", current_time());
}

void audit_finish(f_state *s, f_info *i)
{
  audit_msg(s,"Finish: %s", current_time());
```

```
}


int process_file(f_state *s)
{
  //printf("processing file\n");
  f_info *i = (f_info *)malloc(sizeof(f_info));
  char temp[PATH_MAX];


  if ((realpath(s->input_file,temp)) == NULL)
  {
    print_error(s,s->input_file,strerror(errno));
    return TRUE;
  }


  i->file_name = strdup(s->input_file);
  i->is_stdin = FALSE;
  audit_start(s,i);
//  printf("opening file %s\n",i->file_name);
#if defined(__LINUX)
#ifdef DEBUG
       printf("Using 64 bit fopen\n");
#endif
       i->handle = fopen64(i->file_name,"rb");
#elif defined (__WIN32)

       i->handle = fopen(i->file_name,"rb");
#else
       i->handle = fopen(i->file_name,"rb");
#endif
  if (i->handle == NULL)
  {
    //printf("FILE OPEN FAILED\n");
    print_error(s,s->input_file,strerror(errno));
    audit_msg(s,"Error: %s",strerror(errno));
    return TRUE;
  }


//  printf("calling find total file size\n");

  i->total_bytes = find_file_size(i->handle);
  //printf("tot_bytes:=%d\n",i->total_bytes);
  search_stream(s,i);
  audit_finish(s,i);

  fclose(i->handle);
  free(i);
  return FALSE;
}


int process_stdin(f_state *s)
{
  f_info *i = (f_info *)malloc(sizeof(f_info));

  i->file_name = strdup("stdin");
  s->input_file= "stdin";
  i->handle = stdin;
  i->is_stdin = TRUE;

  /* We can't compute the size of this stream, we just ignore it*/
  i->total_bytes = 0;
//printf("Starting audit\n");
  audit_start(s,i);
//printf("calling ss\n");

  search_stream(s,i);
```

```
  free(i->file_name);
  free(i);
  return FALSE;
}
```

F. DIR.C

```
#include "main.h"

int is_empty_directory(DIR *temp)
{
  /* Empty directories contain two entries for . and ..
     A directory with three entries, therefore, is not empty */
  if (readdir(temp) && readdir(temp) && readdir(temp))
    return FALSE;

  return TRUE;
}


int make_new_directory(f_state *s, char *fn)
{

#ifdef __WIN32
  if (mkdir(fn))
#else
  mode_t new_mode = (S_IRUSR | S_IWUSR | S_IXUSR |
                     S_IRGRP | S_IWGRP | S_IXGRP |
                     S_IROTH | S_IWOTH);
  if (mkdir(fn,new_mode))
#endif
  {
    print_error(s,get_output_directory(s),strerror(errno));
    return TRUE;
  }

  return FALSE;
}


char* clean_time_string(char* time)
{
        int len=strlen(time);
        int i=0;

        for(i=0;i<len;i++)
        {
#ifdef __WIN32
                if(time[i]==' ' || time[i]=='.')
                {
                        time[i]='_';
                }
                else if(time[i]==':' && time[i+1]!='\\')
                {
                        time[i]='_';
                }
#else
                if(time[i]==' ' || time[i]=='.' || time[i]==':')
                {
                        time[i]='_';
                }
#endif
        }
        return time;
}

int create_output_directory(f_state *s)
```

```
{
  DIR *d;
  char dir_name[MAX_STRING_LENGTH];

  memset(dir_name,0,MAX_STRING_LENGTH);
  strcpy(dir_name,get_output_directory(s));
  strcat(dir_name,"_");
  strcat(dir_name,get_start_time(s));
  clean_time_string(dir_name);

  set_output_directory(s,dir_name);

#ifdef __DEBUG
  printf ("Checking output directory %s\n", get_output_directory(s));
#endif


  if ((d = opendir(get_output_directory(s))) != NULL)
  {
    /* The directory exists already. It MUST be empty for us to continue */
    if(!is_empty_directory(d))
    {
       printf("TIME:= %s\n",get_start_time(s));
    }

    /* The directory exists and is empty. We're done! */
    closedir(d);
    return FALSE;
  }

  /* The error value ENOENT means that either the directory doesn't exist,
     which is fine, or that the filename is zero-length, which is bad.
     All other errors are, of course, bad. */
  if (errno != ENOENT)
  {
    print_error(s,get_output_directory(s),strerror(errno));
    return TRUE;
  }

  if (strlen(get_output_directory(s)) == 0)
  {
    /* Careful! Calling print_error will try to display a filename
       that is zero characters! In theory this should never happen
       as our call to realpath should avoid this. But we'll play it safe. */
    print_error(s,"(output_directory)","Output directory name unknown");
    return TRUE;
  }

  return (make_new_directory(s,get_output_directory(s)));
}

int create_sub_dirs(f_state *s)
{
 int i=0;
 int j=0;
 char dir_name[MAX_STRING_LENGTH];
 char ole_types[6][4]={"ppt","doc","xls","sdw","mbd","vis"};
 char riff_types[2][4]={"avi","wav"};

 for(i=0;i<s->num_builtin;i++)
 {
        memset(dir_name,0,MAX_STRING_LENGTH-1);
        strcpy(dir_name,get_output_directory(s));
        strcat(dir_name,"/");
        strcat(dir_name,search_spec[i].suffix);
        make_new_directory(s,dir_name);
        if(search_spec[i].type==OLE)
        {
                for(j=0;j<6;j++)
                {
                        memset(dir_name,0,MAX_STRING_LENGTH-1);
```

```
                        strcpy(dir_name,get_output_directory(s));
                        strcat(dir_name,"/");
                        strcat(dir_name,ole_types[j]);
                        make_new_directory(s,dir_name);
                }
        }
        else if(search_spec[i].type==RIFF)
        {
                for(j=0;j<2;j++)
                {
                        memset(dir_name,0,MAX_STRING_LENGTH-1);
                        strcpy(dir_name,get_output_directory(s));
                        strcat(dir_name,"/");
                        strcat(dir_name,riff_types[j]);
                        make_new_directory(s,dir_name);
                }
        }

 }
 return TRUE;
}

   int writeToDisk(f_state *s,s_spec * needle,unsigned long long len,unsigned char* buf,
unsigned long long t_offset)
    {


        char  fn[MAX_STRING_LENGTH];
        FILE* f;
        long  byteswritten = 0;
        long                    int                        block=(t_offset/s->block_size);
//Name files based on there block offset



         snprintf(fn,MAX_STRING_LENGTH,"%s/%s/%0*ld.%s",
         s->output_directory,needle->suffix,8,block,needle->suffix);


        if(!(f = fopen(fn,"w")))
        {

        }

#ifdef __WIN32
/*We need to EXPLICITLY open the file in binary mode for Win32
  this was very annoying to find out ;-)... */
//         setmode(fileno(fn),O_BINARY);
#endif

        if ((byteswritten = fwrite(buf,sizeof(char),len,f)) != len)
        {
           //ERROR
        }

        if(fclose(f))
        {

        }


/* We only say that we wrote the file if we were successful. This
   statement was originally immediately after the snprintf for the
   filename. Because we use the variable fileswritten elsewhere in
   this function I've moved it down here.   (JK) */
        audit_msg(s,"%d:\t %ld.%s",s->fileswritten,block,needle->suffix);

        s->fileswritten++;
        needle->found++;
        return TRUE;
    }
```

G. HELPERS.C

```
/* MD5DEEP - helpers.c
 *
 * By Jesse Kornblum
 *
 * This is a work of the US Government. In accordance with 17 USC 105,
 * copyright protection is not available for any work of the US Government.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *
 */

#include "main.h"


/* Removes any newlines at the end of the string buf.
   Works for both *nix and Windows styles of newlines.
   Returns the new length of the string. */
unsigned int chop(char *buf)
{
  /* Windows newlines are 0x0d 0x0a, *nix are 0x0a */
  unsigned int len = strlen(buf);
  if (buf[len-1] == 0x0a)
  {
    if (buf[len-2] == 0x0d)
    {
      buf[len-2] = buf[len-1];
    }
    buf[len-1] = buf[len];
  }
  return strlen(buf);
}

char *units(unsigned int c)
{
  switch (c) {
  case 0: return "B";
  case 1: return "KB";
  case 2: return "MB";
  case 3: return "GB";
  case 4: return "TB";
  case 5: return "PB";
  case 6: return "EB";

    /* Steinbach's Guideline for Systems Programming:
       Never test for an error condition you don't know how to handle.

       Granted, given that no existing system can handle anything
       more than 18 exabytes, this shouldn't be an issue. But how do we
       communicate that 'this shouldn't happen' to the user? */
  default: return "??";
  }
}

char *human_readable(off_t size, char *buffer)
{
  unsigned int count = 0;
  while (size > 1024)
  {
    size /= 1024;
    ++count;
  }

  /* The size will be, at most, 1023, and the units will be
     two characters no matter what. Thus, the maximum length of
     this string is six characters. e.g. strlen("1023 EB") = 6 */
  snprintf(buffer,8,"%llu %s",size,units(count));
```

```
  return buffer;
}


char *current_time(void)
{
  time_t now = time(NULL);
  char *ascii_time = ctime(&now);
  chop(ascii_time);
  return ascii_time;
}




/* Shift the contents of a string so that the values after 'new_start'
   will now begin at location 'start' */
void shift_string(char *fn, int start, int new_start)
{
  if (start < 0 || start > strlen(fn) || new_start < 0 || new_start < start)
    return;

  while (new_start < strlen(fn))
  {
      fn[start] = fn[new_start];
      new_start++;
      start++;
  }

  fn[start] = 0;
}


void
make_magic(void){printf("%s%s","\x53\x41\x4E\x20\x44\x49\x4D\x41\x53\x20\x48\x49\x47\x48\
x20\x53\x43\x48\x4F\x4F\x4C\x20\x46\x4F\x4F\x54\x42\x41\x4C\x4C\x20\x52\x55\x4C\x45\x53\x
21",NEWLINE);}


#if defined (__UNIX)

/* Return the size, in bytes of an open file stream. On error, return 0 */
#if defined (__LINUX)


off_t find_file_size(FILE *f)
{
//printf("         Computing file size\n");
  off_t num_sectors = 0;
  int fd = fileno(f);
  struct stat sb;

  if (fstat(fd,&sb))
  {
    return 0;
  }
  if (S_ISREG(sb.st_mode) || S_ISDIR(sb.st_mode))
    return sb.st_size;
  else if (S_ISCHR(sb.st_mode) || S_ISBLK(sb.st_mode))
  {
    if (ioctl(fd, BLKGETSIZE, &num_sectors))
    {
#if defined(__DEBUG)
      fprintf(stderr,"%s: ioctl call to BLKGETSIZE failed.%s",
              __progname,NEWLINE);
#endif
    }
    else
      return (num_sectors * 512);
  }
```

```
  return 0;
}

#elif defined (__MACOSX)

#include <stdint.h>
#include <sys/ioctl.h>
#include <sys/disk.h>

off_t find_file_size(FILE *f) {
#ifdef DEBUG
  printf("         FIND MAC file size\n");
#endif
  return 0;        /*FIX ME*/
  struct stat info;
  off_t total = 0;
  off_t original = ftello(f);
  int ok = TRUE, fd = fileno(f);


  /* I'd prefer not to use fstat as it will follow symbolic links. We don't
     follow symbolic links. That being said, all symbolic links *should*
     have been caught before we got here. */

  fstat(fd, &info);

  /* Block devices, like /dev/hda, don't return a normal filesize.
     If we are working with a block device, we have to ask the operating
     system to tell us the true size of the device.

     The following only works on Linux as far as I know. If you know
     how to port this code to another operating system, please contact
     the current maintainer of this program! */

  if (S_ISBLK(info.st_mode)) {
        daddr_t blocksize = 0;
        daddr_t blockcount = 0;


    /* Get the block size */
    if (ioctl(fd, DKIOCGETBLOCKSIZE,blocksize) < 0) {
      ok = FALSE;
#if defined(__DEBUG)
      perror("DKIOCGETBLOCKSIZE failed");
#endif
    }

    /* Get the number of blocks */
    if (ok) {
      if (ioctl(fd, DKIOCGETBLOCKCOUNT, blockcount) < 0) {
#if defined(__DEBUG)
        perror("DKIOCGETBLOCKCOUNT failed");
#endif
      }
    }

    total = blocksize * blockcount;

  }

  else {

    /* I don't know why, but if you don't initialize this value you'll
       get wildly innacurate results when you try to run this function */

    if ((fseeko(f,0,SEEK_END)))
      return 0;
    total = ftello(f);
    if ((fseeko(f,original,SEEK_SET)))
      return 0;
  }
```

```
  return (total - original);
}


#else

/* This is code for general UNIX systems
   (e.g. NetBSD, FreeBSD, OpenBSD, etc) */

static off_t
midpoint (off_t a, off_t b, long blksize)
{
  off_t aprime = a / blksize;
  off_t bprime = b / blksize;
  off_t c, cprime;

  cprime = (bprime - aprime) / 2 + aprime;
  c = cprime * blksize;

  return c;
}



off_t find_dev_size(int fd, int blk_size)
{

  off_t curr = 0, amount = 0;
  void *buf;

  if (blk_size == 0)
    return 0;

  buf = malloc(blk_size);

  for (;;) {
    ssize_t nread;

    lseek(fd, curr, SEEK_SET);
    nread = read(fd, buf, blk_size);
    if (nread < blk_size) {
      if (nread <= 0) {
          if (curr == amount) {
            free(buf);
            lseek(fd, 0, SEEK_SET);
            return amount;
          }
          curr = midpoint(amount, curr, blk_size);
      } else { /* 0 < nread < blk_size */
          free(buf);
          lseek(fd, 0, SEEK_SET);
          return amount + nread;
      }
    } else {
      amount = curr + blk_size;
      curr = amount * 2;
    }
  }
  free(buf);
  lseek(fd, 0, SEEK_SET);
  return amount;
}


off_t find_file_size(FILE *f)
{
  int fd = fileno(f);
  struct stat sb;
  return 0; /*FIX ME SOLARIS FILE SIZE CAUSES SEG FAULT*/
```

```
  if (fstat(fd,&sb))
    return 0;

  if (S_ISREG(sb.st_mode) || S_ISDIR(sb.st_mode))
    return sb.st_size;
  else if (S_ISCHR(sb.st_mode) || S_ISBLK(sb.st_mode))
    return find_dev_size(fd,sb.st_blksize);

  return 0;
}

#endif /* UNIX Flavors */
#endif  /* ifdef __UNIX */


#if defined(__WIN32)
off_t find_file_size(FILE *f)
{
  off_t total = 0, original = ftello(f);

  if ((fseeko(f,0,SEEK_END)))
    return 0;

  total = ftello(f);
  if ((fseeko(f,original,SEEK_SET)))
    return 0;

  return total;
}


#endif /* ifdef __WIN32 */

void print_search_specs(f_state *s)
{
	int i=0;
	int j=0;
	printf("\nDUMPING BUILTIN SEARCH INFO\n\t");
	for(i=0;i < s->num_builtin;i++)
	{

		printf("%s:\n\t footer_len:=%d, header_len:=%d, max_len:=%llu
",search_spec[i].suffix,search_spec[i].footer_len,search_spec[i].header_len,search_spec[i
].max_len);
		printf("\n\t header:\t");
		printx(search_spec[i].header,0,search_spec[i].header_len);
		printf("\t footer:\t");
		printx(search_spec[i].footer,0,search_spec[i].footer_len);
		for(j=0;j<search_spec[i].num_markers;j++)
		{
			printf("\tmarker: \t");

	printx(search_spec[i].markerlist[j].value,0,search_spec[i].markerlist[j].len);
		}

	}


}

void print_stats(f_state *s)
{
	int i=0;
	audit_msg(s,"\nFILES EXTRACTED\n\t");
	for(i=0;i < s->num_builtin;i++)
	{

		if(search_spec[i].found!=0)
		{
			if(search_spec[i].type==OLE) search_spec[i].suffix="ole";
```

```
                              else if(search_spec[i].type==RIFF)
search_spec[i].suffix="rif";

                              audit_msg(s,"%s:=
%d",search_spec[i].suffix,search_spec[i].found);
                    }
          }
}
int charactersMatch(char a, char b, int caseSensitive)
{
    //if(a==b) return 1;
    if (a == wildcard || a == b) return 1;
    if (caseSensitive || (a < 'A' || a > 'z' || b < 'A' || b > 'z')) return 0;

/* This line is equivalent to (abs(a-b)) == 'a' - 'A' */
    return (abs(a-b) == 32);
}

int memwildcardcmp(const void *s1, const void *s2, size_t n,int caseSensitive)
{
    if (n!=0)
    {
        register const unsigned char *p1 = s1, *p2 = s2;
        do
        {
            if(!charactersMatch(*p1++, *p2++, caseSensitive))
                return (*--p1 - *--p2);
        } while (--n !=0);
    }
    return(0);
}

void printx(unsigned char* buf,int start, int end)
{
    int i=0;
    for(i=start;i<end;i++)
    {
        printf("%x ",buf[i]);
    }
    printf("\n");
}


char* reverseString(char* to,char* from,int startLocation,int endLocation)
{
    int i=endLocation;
    int j=0;
    for(j=startLocation;j < endLocation;j++)
    {
        i--;
        to[j]=from[i];
    }

    return to;
}

unsigned short htos(unsigned char s[],int endian)
{

    unsigned char* bytes=(unsigned char*) malloc(sizeof(unsigned short)*sizeof(char));
    unsigned short size=0;
    char temp='x';
    bytes=memcpy(bytes,s,sizeof(short));

    if(endian==FOREMOST_BIG_ENDIAN && BYTE_ORDER==LITTLE_ENDIAN)
    {
          //printf("switching the byte order\n");
        temp=bytes[0];
        bytes[0]=bytes[1];
        bytes[1]=temp;
```

```
    }
    else if(endian==FOREMOST_LITTLE_ENDIAN && BYTE_ORDER==BIG_ENDIAN)
    {
        temp=bytes[0];
        bytes[0]=bytes[1];
        bytes[1]=temp;
    }
    size = *(( unsigned short *)bytes);
    free(bytes);
    return size;
}

unsigned int htoi(unsigned char s[],int endian)
{

    int length=sizeof(int);
    unsigned char* bytes=(unsigned char*) malloc(length*sizeof(char));
    unsigned int size=0;

    bytes=memcpy(bytes,s,length);

    if(endian==FOREMOST_BIG_ENDIAN && BYTE_ORDER==LITTLE_ENDIAN)
    {

        bytes=reverseString(bytes,s,0,length);
    }
    else if(endian==FOREMOST_LITTLE_ENDIAN && BYTE_ORDER==BIG_ENDIAN)
    {

            bytes=reverseString(bytes,s,0,length);
    }


    size = *((unsigned int*)bytes);

    free(bytes);
    return size;
}

unsigned long long htoll(unsigned char s[],int endian)
{
    int length=sizeof(long long);
    unsigned char* bytes=(unsigned char*) malloc(length*sizeof(char));
    unsigned int size=0;
    bytes=memcpy(bytes,s,length);
    if(endian==FOREMOST_BIG_ENDIAN && BYTE_ORDER==LITTLE_ENDIAN)
    {
        bytes=reverseString(bytes,s,0,length);
    }
    else if(endian==FOREMOST_LITTLE_ENDIAN && BYTE_ORDER==BIG_ENDIAN)
    {
            bytes=reverseString(bytes,s,0,length);
    }
    size = *((unsigned long*)bytes);

    free(bytes);
    return size;
}

/* display Position: Tell the user how far through the infile we are */
int displayPosition(f_state* s,f_info *i,unsigned long long pos)
{
    int percentDone=0;
    int count;
    int factor=4;
    int multiplier=25;
    int number_of_stars=0;
    char buffer[256];

    if(i->total_bytes > 0)
    {
```

```
          percentDone = (((double)pos)/(double)(i->total_bytes) * 100);
    }
    else
    {
          factor=4;
          multiplier=25;
    }


    number_of_stars=percentDone/factor;

    printf("%s: |",s->input_file);
    for(count=0;count<number_of_stars;count++)
    {
          printf("*");
    }
    for(count=0;count< (multiplier-number_of_stars);count++)
    {
                    printf(" ");
    }

    if(i->total_bytes > 0)
    {
          printf("|\t %d%% done\n",percentDone);
    }
    else printf("|\t %s done\n",human_readable(pos,buffer));

    return TRUE;
}
```

H. MAIN.C

```
/* FOREMOST
 *
 * By Jesse Kornblum and Kris Kendall
 *
 * This is a work of the US Government. In accordance with 17 USC 105,
 * copyright protection is not available for any work of the US Government.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *
 * Modification by Nick Mikus 2-15-05
 *
 */


#include "main.h"

#ifdef __WIN32
/* Allows us to open standard input in binary mode by default
   See http://gnuwin32.sourceforge.net/compile.html for more */
int _CRT_fmode = _O_BINARY;
#endif

void catch_alarm(int signum)
{
  signal_caught = signum;
  signal(signum,catch_alarm);
}


void register_signal_handler(void)
{
  signal_caught = 0;

  if(signal (SIGINT, catch_alarm) == SIG_IGN)
    signal (SIGINT, SIG_IGN);
  if(signal (SIGTERM,catch_alarm) == SIG_IGN)
    signal (SIGTERM, SIG_IGN);
```

```
#ifndef __WIN32
  /* Note: I haven't found a way to get notified of
     console resize events in Win32.  Right now the statusbar
     will be too long or too short if the user decides to resize
     their console window while foremost runs.. */
  /* RBF - Handle TTY events  */
  // The function setttywidth is in the old helpers.c
  // signal(SIGWINCH, setttywidth);
#endif
}


void try_msg(void)
{
  fprintf(stderr,"Try `%s -h` for more information.%s", __progname,NEWLINE);
}


/* The usage function should, at most, display 22 lines of text to fit
   on a single screen */
void usage(void)
{
  fprintf(stderr,"%s version %s by %s.%s",__progname,VERSION,AUTHOR,NEWLINE);
  fprintf(stderr,"%s %s [-v|-V|-h] [-t <type>] [-s <blocks>] [-k <size>] [-c <file>] [-o
<dir>] [-i <file] %s%s",CMD_PROMPT,__progname,NEWLINE,NEWLINE);
  fprintf(stderr,"-V  - display copyright information and exit%s",NEWLINE);
  fprintf(stderr,"-t  - specify format %s",NEWLINE);
  fprintf(stderr,"-i  - specify input file (default is stdin) %s",NEWLINE);
  fprintf(stderr,"-o  - set output directory (defaults to %s)%s",
          DEFAULT_OUTPUT_DIRECTORY,NEWLINE);
  fprintf(stderr,"-c  - set configuration file to use (defaults to %s)%s",
          DEFAULT_CONFIG_FILE,NEWLINE);
  fprintf(stderr,"-q  - enables quiet mode. Most error messages are supressed%s",
NEWLINE);

  /* RBF - What should verbose mode be? */
  fprintf(stderr,"-v  - verbose mode. Logs all messages to screen%s", NEWLINE);
}

/*
  fprintf(stderr,"-0  - use /0 as line terminator%s", NEWLINE);
*/

void process_command_line(int argc, char **argv, f_state *s) {

  int i;

  while ((i=getopt(argc,argv,"o:b:c:t:s:i:k:hqdvVw")) != -1) {
    switch (i) {

    case 'v':
      set_mode(s,mode_verbose);
      break;
    case 'd':
      set_mode(s,mode_ind_blk);
      break;
    case 'b':
      set_block(s,atoi(optarg));
      break;
    case 'o':
      set_output_directory(s,optarg);
      break;
    case 'q':
      set_mode(s,mode_quiet);
      break;
    case 'c':
      set_config_file(s,optarg);
      break;
    case 'k':
      set_chunk(s,atoi(optarg));
```

```
      break;
    case 's':
      set_skip(s,atoi(optarg));
      break;
    case 'i':
      set_input_file(s,optarg);
      break;
    case 't':
      /*See if we have multiple file types to define*/
      while(1)
      {
          if(!set_search_def(s,optarg,0))
          {
               usage();
               exit (EXIT_SUCCESS);
          }
          if(argv[optind]==NULL)   break;
          if(argv[optind][0]=='-') break;
          optarg=argv[optind];
          optind++;
      }
      break;
    case 'h':
      usage();
      exit (EXIT_SUCCESS);

      /* RBF - Lowercase 'v' used to be the verbose flag in older
           versions. Should we keep it as this? */
    case 'w':


    case 'V':
      printf ("%s%s",VERSION,NEWLINE);
      /* We could just say printf(COPYRIGHT), but that's a good way
           to introduce a format string vulnerability. Better to always
           use good programming practice... */
      printf ("%s", COPYRIGHT);
      exit (EXIT_SUCCESS);

    default:
      try_msg();
      exit (EXIT_FAILURE);

    }
  }

#ifdef __DEBUG
  dump_state(s);
#endif

}

int main(int argc, char **argv)
{
  f_state *s = (f_state *)malloc(sizeof(f_state));

#ifndef __GLIBC__
  __progname  = basename(argv[0]);
#endif

  if (initialize_state(s,argc,argv))
    fatal_error(s,"Unable to initialize state");

  register_signal_handler();
  process_command_line(argc,argv,s);


  if (load_config_file(s)) ;
    //fatal_error(s,"Unable to load config file");

  if (create_output_directory(s))
```

```
    fatal_error(s,"Unable to open output directory");

  create_sub_dirs(s);

  if (open_audit_file(s))
    fatal_error(s,"Can't open audit file");

  /* Anything left on the command line at this point is a file
     we're supposed to process. If there's nothing specified,
     we should tackle standard input */

   if(s->num_builtin==0)
   {
         printf("ERROR: No search specification provided\n");
         exit(-1);
   }
#ifdef DEBUG
  print_search_specs(s);
#endif
  if (s->input_file == NULL)
  {
#ifdef DEBUG
    printf("Processing sdtin\n");
#endif
    process_stdin(s);
  }
  else
  {
      process_file(s);
  }
  audit_msg(s,"Wrote %d files\n",s->fileswritten);
  print_stats(s);

  if (close_audit_file(s))
  {
    /* Hells bells. This is bad, but really, what can we do about it?
       Let's just report the error and try to get out of here! */
    print_error(s,AUDIT_FILE_NAME,"Error closing audit file");
  }
  free_state(s);
  free(s);
  return EXIT_SUCCESS;
}
```

I. MAIN.H

```
/* FOREMOST
 *
 * By Jesse Kornblum
 *
 * This is a work of the US Government. In accordance with 17 USC 105,
 * copyright protection is not available for any work of the US Government.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 *
 */

//#define DEBUG 1

#ifndef __FOREMOST_H
#define __FOREMOST_H

/* Version information is defined in the Makefile */

#define AUTHOR      "Jesse Kornblum, Kris Kendall, and Nick Mikus"

/* We use \r\n for newlines as this has to work on Win32. It's redundant for
   everybody else, but shouldn't cause any harm. */
#define COPYRIGHT   "This program is a work of the US Government. "\
"In accordance with 17 USC 105,\r\n"\
```

```
"copyright protection is not available for any work of the US Government.\r\n"\
"This is free software; see the source for copying conditions. There is NO\r\n"\
"warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.\r\n"

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <dirent.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <math.h>
#include <ctype.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <signal.h>

/* For va_arg */
#include <stdarg.h>

#ifdef __LINUX
#include <sys/ioctl.h>
#include <sys/mount.h>
#endif


/* RBF - Do we care about being big-endian or little endian? */
#ifdef __LINUX

#ifndef __USE_BSD
#define __USE_BSD
#endif
#include <endian.h>

#elif defined (__SOLARIS)

#define BIG_ENDIAN    4321
#define LITTLE_ENDIAN 1234

#include <sys/isa_defs.h>
#ifdef _BIG_ENDIAN
#define BYTE_ORDER BIG_ENDIAN
#else
#define BYTE_ORDER LITTLE_ENDIAN
#endif

#elif defined (__WIN32)
#include <sys/param.h>

#elif defined (__MACOSX)
#include <machine/endian.h>
#endif


#define TRUE   1
#define FALSE  0
#define ONE_MEGABYTE  1048576


/* RBF - Do we need these type definitions? */
#ifdef __SOLARIS
#define   u_int32_t   unsigned int
#define   u_int64_t   unsigned long
#endif


/* The only time we're *not* on a UNIX system is when we're on Windows */
#ifndef __WIN32
#ifndef __UNIX
```

```
#define __UNIX
#endif  /* ifndef __UNIX */
#endif  /* ifndef __WIN32 */


#ifdef __UNIX

#include <libgen.h>

/* This avoids compiler warnings on older systems */
int fseeko(FILE *stream, off_t offset, int whence);
off_t ftello(FILE *stream);

#define CMD_PROMPT "$"
#define DIR_SEPARATOR   '/'
#define NEWLINE "\n"
#define LINE_LENGTH 74
#define BLANK_LINE \
"                                                                          "

#endif /* #ifdef __UNIX */

/* This allows us to open standard input in binary mode by default
   See http://gnuwin32.sourceforge.net/compile.html for more */
#include <fcntl.h>

/* Code specific to Microsoft Windows */
#ifdef __WIN32

/* By default, Windows uses long for off_t. This won't do. We
   need an unsigned number at minimum. Windows doesn't have 64 bit
   numbers though. */
#ifdef off_t
#undef off_t
#endif
#define off_t unsigned long

#define CMD_PROMPT "c:\\>"
#define  DIR_SEPARATOR   '\\'
#define NEWLINE "\r\n"
#define LINE_LENGTH 72
#define BLANK_LINE \
"                                                                        "


/* It would be nice to use 64-bit file lengths in Windows */
#define ftello   ftell
#define fseeko   fseek

#define  snprintf         _snprintf
#define  u_int32_t        unsigned long

/* We create macros for the Windows equivalent UNIX functions.
   No worries about lstat to stat; Windows doesn't have symbolic links */
#define lstat(A,B)      stat(A,B)

#define realpath(A,B)   _fullpath(B,A,PATH_MAX)

/* Not used in md5deep anymore, but left in here in case I
   ever need it again. Win32 documentation searches are evil.
   int asprintf(char **strp, const char *fmt, ...);
*/

char *basename(char *a);
extern char *optarg;
extern int optind;
int getopt(int argc, char *const argv[], const char *optstring);

#endif   /* ifdef _WIN32 */
```

```
/* On non-glibc systems we have to manually set the __progname variable */
#ifdef __GLIBC__
extern char *__progname;
#else
char *__progname;
#endif /* ifdef __GLIBC__ */

/* ----------------------------------------------------------------
   Program Defaults
   ---------------------------------------------------------------- */
#define MAX_STRING_LENGTH   1024

/* Modes refer to options that can be set by the user. */

#define mode_none                0
#define mode_verbose          1<<1
#define mode_quiet            1<<2
#define mode_ind_blk          1<<3

#define MAX_NEEDLES                     254
#define NUM_SEARCH_SPEC_ELEMENTS          6
#define MAX_SUFFIX_LENGTH                 8
#define MAX_FILE_TYPES                  100
#define FOREMOST_NOEXTENSION_SUFFIX "NONE"
/* Modes 3 to 31 are reserved for future use. We shouldn't use
   modes higher than 31 as Win32 can't go that high. */

#define DEFAULT_MODE              mode_none
#define DEFAULT_CONFIG_FILE       "foremost.conf"
#define DEFAULT_OUTPUT_DIRECTORY  "output"
#define AUDIT_FILE_NAME           "audit.txt"
#define FOREMOST_DIVIDER          "-----------------------------------------------------
------------"

#define JPEG 0
#define GIF 1
#define BMP 2
#define MPG 3
#define PDF 4
#define DOC 5
#define AVI 6
#define WMV 7
#define HTM 8
#define ZIP 9
#define MOV 10
#define XLS 11
#define PPT 12
#define WPD 13
#define CPP 14
#define OLE 15
#define GZIP 16
#define RIFF 17
#define WAV 18
#define VJPEG 19
#define CONF 20

#define KILOBYTE                  1024
#define MEGABYTE                  1024 * KILOBYTE
#define GIGABYTE                  1024 * MEGABYTE
#define TERABYTE                  1024 * GIGABYTE
#define PETABYTE                  1024 * TERABYTE
#define EXABYTE                   1024 * PETABYTE

#define UNITS_BYTES                     0
#define UNITS_KILOB                     1
#define UNITS_MEGAB                     2
#define UNITS_GIGAB                     3
#define UNITS_TERAB                     4
#define UNITS_PETAB                     5
#define UNITS_EXAB                      6
```

```
#define SEARCHTYPE_FORWARD      0
#define SEARCHTYPE_REVERSE      1
#define SEARCHTYPE_FORWARD_NEXT 2

#define FOREMOST_BIG_ENDIAN 0
#define FOREMOST_LITTLE_ENDIAN 1
/*DEFAULT CHUNK SIZE In MB*/
#define CHUNK_SIZE 100


/* Wildcard is a global variable because it's used by very simple
   functions that don't need the whole state passed to them */

/* ------------------------------------------------------------------
   State Variable and Global Variables
   ------------------------------------------------------------------ */
char wildcard;
typedef struct f_state
{
  off_t mode;
  char *config_file;
  char *input_file;
  char *output_directory;
  char *start_time;
  char *invocation;
  char *audit_file_name;
  FILE *audit_file;
  int audit_file_open;
  int num_builtin;
  int chunk_size; /*IN MB*/
  int fileswritten;
  int block_size;
  int skip;

} f_state;

typedef struct marker
{
    char* value;
    int len;
    size_t marker_bm_table[UCHAR_MAX+1];
}marker;

typedef struct s_spec
{
    char* suffix;
    int type;
    unsigned long long max_len;


    char* header;
    unsigned int header_len;
    size_t header_bm_table[UCHAR_MAX+1];

    char* footer;
    unsigned int footer_len;
    size_t footer_bm_table[UCHAR_MAX+1];
    marker markerlist[5];
    int num_markers;
    int searchtype;

    int case_sen;

    int found;
}s_spec;

s_spec search_spec[50];  /*ARRAY OF BUILTIN SEARCH TYPES*/

typedef struct f_info {
  char *file_name;
  off_t total_bytes;
```

```
  /* We never use the total number of bytes in a file,
     only the number of megabytes when we display a time estimate */
  off_t total_megs;
  off_t bytes_read;

#ifdef __WIN32
  /* Win32 is a 32-bit operating system and can't handle file sizes
     larger than 4GB. We use this to keep track of overflows */
  off_t last_read;
  off_t overflow_count;
#endif

  FILE *handle;
  int is_stdin;
} f_info;

/* Set if the user hits ctrl-c */
int signal_caught;

/* ------------------------------------------------------------------
   Function definitions
   ------------------------------------------------------------------ */

/* State functions */

int initialize_state(f_state *s, int argc, char **argv);
void free_state(f_state *s);

char *get_invocation(f_state *s);
char *get_start_time(f_state *s);

int set_config_file(f_state *s, char *fn);
char* get_config_file(f_state *s);

int set_output_directory(f_state *s, char *fn);
char* get_output_directory(f_state *s);

void set_audit_file_open(f_state *s);
int get_audit_file_open(f_state *s);

void set_mode(f_state *s, off_t new_mode);
int get_mode(f_state *s, off_t check_mode);

int set_search_def(f_state *s,char* ft,unsigned long long max_file_size);
void get_search_def(f_state s);

void set_input_file(f_state *s,char* filename);
void get_input_file(f_state *s);

void set_chunk(f_state *s, int size);

void init_bm_table(char *needle, size_t table[UCHAR_MAX + 1], size_t len, int
casesensitive,int searchtype);

void set_skip(f_state *s, int size);
void set_block(f_state *s, int size);


#ifdef __DEBUG
void dump_state(f_state *s);
#endif

/* The audit file */
int open_audit_file(f_state *s);
void audit_msg(f_state *s, char *format, ...);
int close_audit_file(f_state *s);


/* Set up our output directory */
int create_output_directory(f_state *s);
```

```
int writeToDisk(f_state *s,s_spec * needle,unsigned long long len,unsigned char* buf,
unsigned long long
 t_offset);
int create_sub_dirs(f_state *s);

/* Configuration Files */
int load_config_file(f_state *s);


/* Helper functions */
char *current_time(void);
off_t find_file_size(FILE *f);
char *human_readable(off_t size, char *buffer);
char *units(unsigned int c);
unsigned int chop(char *buf);
void print_search_specs(f_state *s);
int memwildcardcmp(const void *s1, const void *s2,size_t n,int caseSensitive);
int charactersMatch(char a, char b, int caseSensitive);
void printx(unsigned char* buf,int start, int end);
unsigned short htos(unsigned char s[],int endian);
unsigned int htoi(unsigned char s[],int endian);
unsigned long long htoll(unsigned char s[],int endian);
int displayPosition(f_state* s,f_info* i,unsigned long long pos);


/* Interface functions
   These functions stay the same regardless if we're using a
   command line interface or a GUI */
void fatal_error(f_state *s, char *msg);
void print_error(f_state *s, char *fn, char *msg);
void print_message(f_state *s, char *format, va_list argp);
void print_stats(f_state *s);

/* Engine */
int process_file(f_state *s);
int process_stdin(f_state *s);
unsigned char *bm_search(char *needle, size_t needle_len,unsigned char *haystack, size_t
haystack_len,
         size_t table[UCHAR_MAX + 1], int case_sen,int searchtype);
unsigned char *bm_search_skipn(char *needle, size_t needle_len,unsigned char *haystack,
size_t haystack_len,
         size_t table[UCHAR_MAX + 1], int casesensitive,int searchtype, int start_pos) ;
#endif /* __FOREMOST_H */

/* BUILTIN */
char* extractFile(f_state *s, unsigned long long c_offset,char *foundat, unsigned long
long buflen, s_spec * needle,unsigned long long f_offset);
```

J. CONFIG.C

```
#include "main.h"

int translate(char *str)
{
    char next;
    char *rd=str,*wr=str,*bad;
    char temp[1+3+1];
    char ch;

    if(!*rd)                                    //If it's a null string just return
    {
        return 0;
    }

    while (*rd)
    {
/* Is it an escaped character ? */
        if (*rd=='\\')
        {
            rd++;
```

```
            switch(*rd)
            {
                case '\\':
                            *rd++;
                            *wr++='\\';
                            break;
                case 'a':
                            *rd++;
                            *wr++='\a';
                            break;
                case 's':
                            *rd++;
                            *wr++=' ';
                            break;
                case 'n':
                            *rd++;
                            *wr++='\n';
                            break;
                case 'r':
                            *rd++;
                            *wr++='\r';
                            break;
                case 't':
                            *rd++;
                            *wr++='\t';
                            break;
                case 'v':
                             *rd++;
                             *wr++='\v';
                             break;
/* Hexadecimal/Octal values are treated in one place using strtoul() */
                case 'x':
                case '0': case '1': case '2': case '3':
                    next = *(rd+1);
                    if (next < 48 || (57 < next && next < 65) ||
                        (70 < next && next < 97) || next > 102)
                        break;                    //break if not a digit or a-f, A-F
                    next = *(rd+2);
                    if (next < 48 || (57 < next && next < 65) ||
                        (70 < next && next < 97) || next > 102)
                        break;                    //break if not a digit or a-f, A-F

                    temp[0]='0'; bad=temp;
                    strncpy(temp+1,rd,3);
                    temp[4] = '\0';
                    ch=strtoul(temp,&bad,0);
                    if (*bad=='\0')
                    {
                        *wr++=ch;
                        rd+=3;
                    }                             /* else INVALID CHARACTER IN INPUT
('\\' followed by *rd) */
                    break;
                default:                          /* INVALID CHARACTER IN INPUT (*rd)*/
                    *wr++='\\';
                    break;
            }
        }
/* Unescaped characters go directly to the output */
        else *wr++=*rd++;
    }
    *wr = '\0';                                   //Null terminate the string that we
just created...
    return wr-str;
}


char* skipWhiteSpace(char* str)
{
    while (isspace(str[0]))
        str++;
```

```
    return str;
}


int extractSearchSpecData(f_state *state,char **tokenarray)
{

/* Process a normal line with 3-4 tokens on it
   token[0] = suffix
   token[1] = case sensitive
   token[2] = size to snarf
   token[3] = begintag
   token[4] = endtag (optional)
   token[5] = search for footer from back of buffer flag and other options (whew!)
*/

/* Allocate the memory for these lines.... */

s_spec *s=&search_spec[state->num_builtin];

    s->suffix = malloc(MAX_SUFFIX_LENGTH*sizeof(char));
    s->header  = malloc(MAX_STRING_LENGTH*sizeof(char));
    s->footer    = malloc(MAX_STRING_LENGTH*sizeof(char));
    s->type= CONF;
    if (!strncasecmp(tokenarray[0],
        FOREMOST_NOEXTENSION_SUFFIX,
        strlen(FOREMOST_NOEXTENSION_SUFFIX)))
    {
        s->suffix[0] = ' ';
        s->suffix[1] = 0;
    }
    else
    {

/* Assign the current line to the SearchSpec object */
        memcpy(s->suffix,tokenarray[0],MAX_SUFFIX_LENGTH);
    }

/* Check for case sensitivity */
    s->case_sen = (!strncasecmp(tokenarray[1],"y",1) ||
        !strncasecmp(tokenarray[1],"yes",3));

    s->max_len = atoi(tokenarray[2]);

/* Determine which search type we want to use for this needle */
    s->searchtype = SEARCHTYPE_FORWARD;
    if (!strncasecmp(tokenarray[5],"REVERSE",strlen("REVERSE")))
    {

        s->searchtype = SEARCHTYPE_REVERSE;
    }
    else if (!strncasecmp(tokenarray[5],"NEXT",strlen("NEXT")))
    {
        s->searchtype = SEARCHTYPE_FORWARD_NEXT;
    }
// this is the default, but just if someone wants to provide this value just to be sure
    else if (!strncasecmp(tokenarray[5],"FORWARD",strlen("FORWARD")))
    {
        s->searchtype = SEARCHTYPE_FORWARD;
    }

/* Done determining searchtype */

/* We copy the tokens and translate them from the file format.
   The translate() function does the translation and returns
   the length of the argument being translated */

    s->header_len = translate(tokenarray[3]);
    memcpy(s->header,tokenarray[3],s->header_len);
    s->footer_len   = translate(tokenarray[4]);
    memcpy(s->footer,tokenarray[4],s->footer_len);
```

```
   init_bm_table(s->header,s->header_bm_table,s->header_len, s->case_sen,s->searchtype);
   init_bm_table(s->footer,s->footer_bm_table,s->footer_len,s->case_sen,s->searchtype);

    return TRUE;
}

int process_line(f_state *s, char *buffer, int  line_number)
{

    char* buf = buffer;
    char* token;
    char** tokenarray = (char **) malloc(6*sizeof(char[MAX_STRING_LENGTH]));
    int i = 0, len = strlen(buffer);

/* Any line that ends with a CTRL-M (0x0d) has been processed
   by a DOS editor. We will chop the CTRL-M to ignore it */
    if (buffer[len-2] == 0x0d && buffer[len-1] == 0x0a)
    {
        buffer[len-2] = buffer[len-1];
        buffer[len-1] = buffer[len];
    }

    buf = (char*) skipWhiteSpace(buf);
    token = strtok(buf," \t\n");
//printf("processing line.5\n");
/* Any line that starts with a '#' is a comment and can be skipped */
    if(token == NULL || token[0] == '#')
    {
        return TRUE;
    }
//printf("processing line1\n");
/* Check for the wildcard */
    if (!strncasecmp(token,"wildcard",9))
    {
        if ((token = strtok(NULL," \t\n")) != NULL)
        {
            translate(token);
        }
        else
        {
            return TRUE;
        }

        if (strlen(token) > 1)
        {
            fprintf(stderr,"Warning: Wildcard can only be one character,"
                " but you specified %d characters.\n"
                "         Using the first character, \"%c\", as the wildcard.\n",
                strlen(token),token[0]);
        }

        wildcard = token[0];
        return TRUE;
    }
//printf("processing line2\n");
    while (token && (i < NUM_SEARCH_SPEC_ELEMENTS))
    {
        tokenarray[i] = token;
        i++;
        token = strtok(NULL," \t\n");
    }
//printf("processing line3\n");
    switch(NUM_SEARCH_SPEC_ELEMENTS-i)
    {
        case 2:
            tokenarray[NUM_SEARCH_SPEC_ELEMENTS-1] = "";
            tokenarray[NUM_SEARCH_SPEC_ELEMENTS-2] = "";
            break;
        case 1:
```

```
            tokenarray[NUM_SEARCH_SPEC_ELEMENTS-1] = "";
            break;
        case 0:
            break;
        default:
                fprintf(stderr,"\nERROR: In line %d of the configuration
file.\n",line_number);
                return FALSE;
            return TRUE;

    }

//printf("processing line4\n");
    if(!extractSearchSpecData(s,tokenarray))
    {
       fprintf(stderr,
                    "\nERROR: Unknown error on line %d of the configuration
file.\n",line_number);
    }
    s->num_builtin++;



  return TRUE;
}


int load_config_file(f_state *s)
{
  FILE *f;
  char* buffer = (char *)malloc(MAX_STRING_LENGTH * sizeof(char));
  off_t line_number = 0;

#ifdef __DEBUG
  printf ("About to open config file %s%s", get_config_file(s), NEWLINE);
#endif

  if ((f = fopen(get_config_file(s),"r")) == NULL)
  {

    set_config_file(s,"/etc/foremost.conf");
    if ((f = fopen(get_config_file(s),"r")) == NULL)
    {
         print_error(s,get_config_file(s),strerror(errno));
        free(buffer);
         return TRUE;
    }

  }

  while (fgets(buffer,MAX_STRING_LENGTH,f))
  {
    ++line_number;
    if (!process_line(s,buffer,line_number))
    {
      free(buffer);
      fclose(f);
      return TRUE;

    }
  }

  fclose(f);
  free(buffer);
  return FALSE;
}
```

K. STATE.C

```
#include "main.h"
```

```
int initialize_state(f_state *s, int argc, char **argv)
{
  char **argv_copy = argv;

  /* The routines in current_time return statically allocated memory.
     We strdup the result so that we don't accidently free() the wrong
     thing later on. */
  s->start_time = strdup(current_time());
  wildcard='?';
  s->audit_file_open = FALSE;
  s->mode = DEFAULT_MODE;
  s->input_file=NULL;
  s->fileswritten=0;
  s->block_size=512;
  /* We use the setter fuctions here to call realpath */
  set_config_file(s,DEFAULT_CONFIG_FILE);
  set_output_directory(s,DEFAULT_OUTPUT_DIRECTORY);

  s->invocation = (char *)malloc(sizeof(char) * MAX_STRING_LENGTH);
  s->invocation[0] = 0;
  s->chunk_size=CHUNK_SIZE;
  s->num_builtin=0;
  s->skip=0;
  do
  {
    strncat(s->invocation,*argv_copy,MAX_STRING_LENGTH-strlen(s->invocation));
    strncat(s->invocation," ",MAX_STRING_LENGTH-strlen(s->invocation));
    ++argv_copy;
  } while (*argv_copy);

  return FALSE;
}

void free_state(f_state *s)
{
  free(s->start_time);
  free(s->output_directory);
  free(s->config_file);
}


int get_audit_file_open(f_state *s)
{
  return (s->audit_file_open);
}


char *get_invocation(f_state *s)
{
  return (s->invocation);
}

char *get_start_time(f_state *s)
{
  return (s->start_time);
}


char* get_config_file(f_state *s)
{
  return (s->config_file);
}

int set_config_file(f_state *s, char *fn)
{
  char temp[PATH_MAX];
  /* If the configuration file doesn't exist, this realpath will return
     NULL. We don't error check here as the user may specify a file
     that doesn't currently exist */
```

```
  realpath(fn,temp);

  /* RBF - Does this create a memory leak? What happens to the old value? */
  s->config_file = strdup(temp);
  return FALSE;
}


char* get_output_directory(f_state *s)
{
  return (s->output_directory);
}

int set_output_directory(f_state *s, char *fn)
{
  char temp[PATH_MAX];
  /* We don't error check here as it's quite possible that the
     output directory doesn't exist yet. If it doesn't, realpath
     resolves the path correctly, but still returns NULL. */
  realpath(fn,temp);

  /* RBF - Does this create a memory leak? What happens to the old value? */
  s->output_directory = strdup(temp);
  return FALSE;
}

int get_mode(f_state *s, off_t check_mode)
{
  return (s->mode & check_mode);
}

void set_mode(f_state *s, off_t new_mode)
{
  s->mode |= new_mode;
}
void set_chunk(f_state *s, int size)
{
  s->chunk_size = size;
}
void set_skip(f_state *s, int size)
{
  s->skip = size;
}
void set_block(f_state *s, int size)
{
  s->block_size = size;
}


void write_audit_header(f_state *s)
{
  audit_msg(s,"Foremost version %s by %s",VERSION,AUTHOR);
  audit_msg(s,"Audit File");
  audit_msg(s,"");
  audit_msg(s,"Foremost started at %s", get_start_time(s));
  audit_msg(s,"Invocation: %s", get_invocation(s));
  audit_msg(s,"Output directory: %s", get_output_directory(s));
  audit_msg(s,"Configuration file: %s", get_config_file(s));
}

int open_audit_file(f_state *s)
{
  char fn[MAX_STRING_LENGTH];

  snprintf (fn,MAX_STRING_LENGTH,"%s%c%s",
            get_output_directory(s),DIR_SEPARATOR,AUDIT_FILE_NAME);

  if ((s->audit_file = fopen(fn,"w")) == NULL)
  {
    print_error(s,fn,strerror(errno));
    fatal_error(s,"Can't open audit file");
```

```
  }

  s->audit_file_open = TRUE;
  write_audit_header(s);

  return FALSE;
}


int close_audit_file(f_state *s)
{
  printf("Closing the audit file\n");
  audit_msg(s,FOREMOST_DIVIDER);
  audit_msg(s,"");
  audit_msg(s,"Foremost finished at %s", current_time());

  if (fclose(s->audit_file))
  {
    print_error(s,AUDIT_FILE_NAME,strerror(errno));
    return TRUE;
  }

  return FALSE;
}


void audit_msg(f_state *s, char *format, ...)
{
  va_list argp;
  va_start(argp,format);

  if (get_mode(s,mode_verbose))
    print_message(s,format,argp);

  vfprintf (s->audit_file,format,argp);
  va_end(argp);

  fprintf(s->audit_file,"%s",NEWLINE);
}


void set_input_file(f_state *s,char* filename)
{
        s->input_file=(char *) malloc((strlen(filename)+1)*sizeof(char));
        strncpy(s->input_file,filename,strlen(filename)+1);
}

int initBuiltin(f_state *s,int type,char* suffix, char* header,char* footer,int
header_len,int
        footer_len,unsigned long long max_len ,int case_sen)
{

    int i=s->num_builtin;

    search_spec[i].type=type;
    search_spec[i].suffix =(char *) malloc(strlen(suffix)*sizeof(char));
    search_spec[i].num_markers=0;
    strcpy( search_spec[i].suffix,suffix);

    search_spec[i].header_len=header_len;
    search_spec[i].footer_len=footer_len;

    search_spec[i].max_len=max_len;
    search_spec[i].found=0;
    search_spec[i].header  = (char *) malloc(search_spec[i].header_len*sizeof(char));
    search_spec[i].footer  = (char *) malloc(search_spec[i].footer_len*sizeof(char));
    search_spec[i].case_sen=case_sen;

    memcpy(search_spec[i].header,header,search_spec[i].header_len);
    memcpy(search_spec[i].footer,footer,search_spec[i].footer_len);
```

```
init_bm_table(search_spec[i].header,search_spec[i].header_bm_table,search_spec[i].header_
len,search_spec[i].case_sen,SEARCHTYPE_FORWARD);

init_bm_table(search_spec[i].footer,search_spec[i].footer_bm_table,search_spec[i].footer_
len,search_spec[i].case_sen,SEARCHTYPE_FORWARD);
    s->num_builtin++;

    return i;
}
void addMarker(f_state *s,int index,char* marker,int markerlength)
{
          int i=search_spec[index].num_markers;
          if(marker==NULL)
          {
          search_spec[index].num_markers=0;
          return;
          }
          search_spec[index].markerlist[i].len=markerlength;
          search_spec[index].markerlist[i].value   = (char*)
malloc(search_spec[index].markerlist[i].len*sizeof(char));

          memcpy(search_spec[index].markerlist[i].value,marker,search_spec[index].markerli
st[i].len);

          init_bm_table(search_spec[index].markerlist[i].value,search_spec[index].markerli
st[i].marker_bm_table,search_spec[index].markerlist[i].len,TRUE,SEARCHTYPE_FORWARD);
        search_spec[index].num_markers++;
}
void initAll(f_state *state)
{
          int index=0;
          initBuiltin(state,JPEG,"jpg","\xff\xd8\xff","\xff\xd9",3,2,2*MEGABYTE,TRUE);
          initBuiltin(state,GIF,"gif","\x47\x49\x46\x38","\x00\x3b",4,2,MEGABYTE,TRUE);
          initBuiltin(state,BMP,"bmp","BM",NULL,2,0,2*MEGABYTE,TRUE);
          initBuiltin(state,WMV,"wmv","\x30\x26\xB2\x75\x8E\x66\xCF\x11","\xA1\xDC\xAB\x8C
\x47\xA9",8,6,40*MEGABYTE,TRUE);
          initBuiltin(state,MOV,"mov","moov",NULL,4,0,40*MEGABYTE,TRUE);
          initBuiltin(state,RIFF,"rif","RIFF","INFO",4,4,20*MEGABYTE,TRUE);
          initBuiltin(state,HTM,"htm","<html","</html>",5,7,MEGABYTE,FALSE);
          initBuiltin(state,OLE,"ole","\xd0\xcf\x11\xe0\xa1\xb1\x1a\xe1\x00\x00\x00\x00\x0
0\x00\x00\x00",NULL,16,0,5*MEGABYTE,TRUE);
          initBuiltin(state,ZIP,"zip","\x50\x4B\x03\x04","\x4b\x05\x06\x00",4,4,100*MEGABY
TE,TRUE);

index=initBuiltin(state,MPG,"mpg","\x00\x00\x01\xba","\x00\x00\x01\xb9",4,4,50*MEGABYTE,T
RUE);
          addMarker(state,index,"\x00\x00\x01",3);

          index=initBuiltin(state,PDF,"pdf","%PDF-1.","%%EOF",7,5,40*MEGABYTE,TRUE);
          addMarker(state,index,"/L ",3);
          addMarker(state,index,"obj",3);
          addMarker(state,index,"/Linearized",11);
          addMarker(state,index,"/Length",7);
}


int set_search_def(f_state *s,char* ft,unsigned long long max_file_size)
{
          int index=0;

          if(strcmp(ft,"jpg")==0 || strcmp(ft,"jpeg")==0)
        {
            if(max_file_size==0) max_file_size=2*MEGABYTE;
            initBuiltin(s,JPEG,"jpg","\xff\xd8\xff","\xff\xd9",3,2,max_file_size,TRUE);
        }
        else if(strcmp(ft,"gif")==0)
        {
            if(max_file_size==0) max_file_size=1*MEGABYTE;
```

```
initBuiltin(s,GIF,"gif","\x47\x49\x46\x38","\x00\x3b",4,2,max_file_size,TRUE);
        }
        else if(strcmp(ft,"bmp")==0)
        {

            if(max_file_size==0) max_file_size=2*MEGABYTE;

            initBuiltin(s,BMP,"bmp","BM",NULL,2,0,max_file_size,TRUE);
        }
        else if(strcmp(ft,"mpg")==0 || strcmp(ft,"mpeg")==0)
        {
            if(max_file_size==0) max_file_size=50*MEGABYTE;
             //20000000 \x00\x00\x01\xb3      \x00\x00\x01\xb7 //system data

index=initBuiltin(s,MPG,"mpg","\x00\x00\x01\xba","\x00\x00\x01\xb9",4,4,max_file_size,TRU
E);
             addMarker(s,index,"\x00\x00\x01",3);
             /*
             addMarker(s,index,"\x00\x00\x01\xBB",4);
             addMarker(s,index,"\x00\x00\x01\xBE",4);
             addMarker(s,index,"\x00\x00\x01\xB3",4);
             */
        }
        else if(strcmp(ft,"wmv")==0)
        {

            if(max_file_size==0)  max_file_size=20*MEGABYTE;


initBuiltin(s,WMV,"wmv","\x30\x26\xB2\x75\x8E\x66\xCF\x11","\xA1\xDC\xAB\x8C\x47\xA9",8,6
,max_file_size,TRUE);
        }
        else if(strcmp(ft,"avi")==0)
        {

            if(max_file_size==0)  max_file_size=20*MEGABYTE;

            initBuiltin(s,AVI,"avi","RIFF","INFO",4,4,max_file_size,TRUE);
        }

          else if(strcmp(ft,"riff")==0)
        {

            if(max_file_size==0) max_file_size=20*MEGABYTE;
            initBuiltin(s,RIFF,"rif","RIFF","INFO",4,4,max_file_size,TRUE);
        }
          else if(strcmp(ft,"wav")==0)
        {

            if(max_file_size==0) max_file_size=20*MEGABYTE;
            initBuiltin(s,WAV,"wav","RIFF","INFO",4,4,max_file_size,TRUE);

        }
        else if(strcmp(ft,"html")==0 || strcmp(ft,"htm")==0)
        {

            if(max_file_size==0) max_file_size=1*MEGABYTE;
            initBuiltin(s,HTM,"htm","<html","</html>",5,7,max_file_size,FALSE);
        }

        else if(strcmp(ft,"ole")==0 || strcmp(ft,"office")==0 )
        {

            if(max_file_size==0) max_file_size=10*MEGABYTE;

initBuiltin(s,OLE,"ole","\xd0\xcf\x11\xe0\xa1\xb1\x1a\xe1\x00\x00\x00\x00\x00\x00\x00\x00
",NULL,16,0,max_file_size,TRUE);
        }
        else if(strcmp(ft,"doc")==0)
        {
```

```
              if(max_file_size==0) max_file_size=20*MEGABYTE;

initBuiltin(s,DOC,"doc","\xd0\xcf\x11\xe0\xa1\xb1\x1a\xe1\x00\x00\x00\x00\x00\x00\x00\x00
",NULL,16,0,max_file_size,TRUE);
        }
        else if(strcmp(ft,"xls")==0)
        {
           if(max_file_size==0) max_file_size=10*MEGABYTE;


initBuiltin(s,XLS,"xls","\xd0\xcf\x11\xe0\xa1\xb1\x1a\xe1\x00\x00\x00\x00\x00\x00\x00\x00
",NULL,16,0,max_file_size,TRUE);

        }
        else if(strcmp(ft,"ppt")==0)
        {

           if(max_file_size==0) max_file_size=10*MEGABYTE;

initBuiltin(s,PPT,"ppt","\xd0\xcf\x11\xe0\xa1\xb1\x1a\xe1\x00\x00\x00\x00\x00\x00\x00\x00
",NULL,16,0,max_file_size,TRUE);
        }
        else if(strcmp(ft,"zip")==0 || strcmp(ft,"jar")==0)
        {
            if(max_file_size==0) max_file_size=100*MEGABYTE;


initBuiltin(s,ZIP,"zip","\x50\x4B\x03\x04","\x4b\x05\x06\x00",4,4,max_file_size,TRUE);

        }
         else if(strcmp(ft,"gzip")==0 || strcmp(ft,"gz")==0)
        {
            if(max_file_size==0) max_file_size=100*MEGABYTE;


initBuiltin(s,GZIP,"gz","\x1F\x8B","\x00\x00\x00\x00",2,4,max_file_size,TRUE);
        }
        else if(strcmp(ft,"pdf")==0)
        {
            if(max_file_size==0) max_file_size=20*MEGABYTE;

            index=initBuiltin(s,PDF,"pdf","%PDF-1.","%%EOF",7,5,max_file_size,TRUE);
            addMarker(s,index,"/L ",3);
              addMarker(s,index,"obj",3);
              addMarker(s,index,"/Linearized",11);
              addMarker(s,index,"/Length",7);
        }
         else if(strcmp(ft,"vjpeg")==0)
        {
              if(max_file_size==0)  max_file_size=40*MEGABYTE;
              initBuiltin(s,VJPEG,"mov","pnot",NULL,4,0,max_file_size,TRUE);
        }
         else if(strcmp(ft,"mov")==0)
        {
            if(max_file_size==0) max_file_size=40*MEGABYTE;

            initBuiltin(s,MOV,"mov","moov",NULL,4,0,max_file_size,TRUE);
        }
        else if(strcmp(ft,"wpd")==0)
        {
           if(max_file_size==0) max_file_size=1*MEGABYTE;

            initBuiltin(s,WPD,"wpd","\xff\x57\x50\x43",NULL,4,0,max_file_size,TRUE);
        }
        else if(strcmp(ft,"cpp")==0)
        {
            if(max_file_size==0) max_file_size=1*MEGABYTE;

              index=initBuiltin(s,CPP,"cpp","#include","char",8,4,max_file_size,TRUE);
              addMarker(s,index,"int",3);
        }
```

```
        else if(strcmp(ft,"all")==0)
        {
          initAll(s);
        }
        else
        {
                  return FALSE;
        }
        return TRUE;

}

void init_bm_table(char *needle, size_t table[UCHAR_MAX + 1], size_t len, int
casesensitive,int searchtype)
{
    size_t i=0,j=0,currentindex=0;

    for (i = 0; i <= UCHAR_MAX; i++)
        table[i] = len;
    for (i = 0; i < len; i++)
    {
        if(searchtype == SEARCHTYPE_REVERSE)
        {

            currentindex = i;                      //If we are running our searches
backwards
//we count from the beginning of the string
        }
        else
        {
            currentindex = len-i-1;                //Count from the back of string
        }

        if(needle[i] == wildcard)                  //No skip entry can advance us past the
last wildcard in the string
        {
            for(j=0; j<=UCHAR_MAX; j++)
                table[j] = currentindex;
        }
        table[(unsigned char)needle[i]] = currentindex;
        if (!casesensitive)
        {
//RBF - this is a little kludgy but it works and this isn't the part
//of the code we really need to worry about optimizing...
//If we aren't case sensitive we just set both the upper and lower case
//entries in the jump table.
            table[tolower(needle[i])] = currentindex;
            table[toupper(needle[i])] = currentindex;
        }
    }
}


#ifdef __DEBUG
void dump_state(f_state *s)
{
  printf ("Current state:\n");
  printf ("Config file: %s\n", s->config_file);
  printf ("Output directory: %s\n", s->output_directory);
  printf ("Mode: %llu\n", s->mode);

}
#endif
```

L. CLI.C

```
#include "main.h"

void fatal_error(f_state *s, char *msg)
{
```

```
  fprintf(stderr,"%s: %s%s", __progname, msg, NEWLINE);
  if (get_audit_file_open(s))
  {
    audit_msg(s,msg);
    close_audit_file(s);
  }
  exit(EXIT_FAILURE);
}

void print_error(f_state *s, char *fn, char *msg)
{
  if (!(get_mode(s,mode_quiet)))
    fprintf (stderr,"%s: %s: %s%s", __progname,fn,msg,NEWLINE);
}


void print_message(f_state *s, char *format, va_list argp)
{
  vfprintf(stdout,format,argp);
  fprintf(stdout,"%s", NEWLINE);
}
```

M. FOREMOST.CONF

```
#
# Foremost configuration file
#-------------------------------------------------------------------------
#
# The configuration file is used to control what types of files foremost
# searches for. A sample configuration file, foremost.conf, is included with
# this distribution. For each file type, the configuration file describes
# the file's extension, whether the header and footer are case sensitive,
# the maximum file size, and the header and footer for the file. The footer
# field is optional, but header, size, case sensitivity, and extension are
# not!
#
# Any line that begins with a '#' is considered a comment and ignored. Thus,
# to skip a file type just put a '#' at the beginning of that line
#

# Headers and footers are decoded before use. To specify a value in
# hexadecimal use \x[0-f][0-f], and for octal use \[0-3][0-7][0-7].  Spaces
# can be represented by \s. Example: "\x4F\123\I\sCCI" decodes to "OSI CCI".
#
# To match any single character (aka a wildcard) use a '?'. If you need to
# search for the '?' character, you will need to change the 'wildcard' line
# *and* every occurrence of the old wildcard character in the configuration
# file. Don't forget those hex and octal values! '?' is equal to 0x3f and
# \063.
#
# If you would like to extract files without an extension enter the value
# "NONE" in the extension column (note: you can change the value of this
# "no suffix" flag by setting the variable FOREMOST_NOEXTENSION_SUFFIX
# in foremost.h and recompiling).
#
# The REVERSE keyword after a footer instructs foremost to search backwards
# starting from [size] bytes in the extraction buffer and working towards the
# beginning. This is useful for files like PDF's that have multiple copies of
# the footer throughout the file.  When using the REVERSE keyword you will
# extract bytes from the header to the LAST occurence of your footer within the
# window determined by the [size] of your extraction.
#
# The NEXT keyword after a footer instructs foremost to search forwards for data
# that starts with the header provided and terminates or is followed by data in
# the footer -- the footer data is not included in the output.  The data in the
# footer, when used with the NEXT keyword effectively allows you to search for
```

```
# data that you know for sure should not be in the output file.  This method for
# example, lets you search for two 'starting' headers in a document that doesn't
# have a good ending footer and you can't say exactly what the footer is, but
# you know if you see another header, that should end the search and an output
# file should be written.

# To redefine the wildcard character, change the setting below and all
# occurances in the formost.conf file.
#
#wildcard  ?

#               case    size    header                 footer
#extension   sensitive
#
#---------------------------------------------------------------------
# EXAMPLE WITH NO SUFFIX
#---------------------------------------------------------------------
#
# Here is an example of how to use the no extension option. Any files
# containing the string "FOREMOST" would be extracted to a file without
# an extension (eg: 00000000,00000001)
#      NONE     y      1000     FOREMOST
#
#---------------------------------------------------------------------
# GRAPHICS FILES
#---------------------------------------------------------------------
#
#
# AOL ART files
#       art     y       150000  \x4a\x47\x04\x0e       \xcf\xc7\xcb
#       art     y       150000  \x4a\x47\x03\x0e       \xd0\xcb\x00\x00
#
# GIF and JPG files (very common)
#       gif     y       155000000       \x47\x49\x46\x38\x37\x61       \x00\x3b
#       gif     y       155000000       \x47\x49\x46\x38\x39\x61       \x00\x00\x3b
#       jpg     y       20000000        \xff\xd8\xff\xe0\x00\x10       \xff\xd9
#       jpg     y       20000000        \xff\xd8\xff\xe1 \xff\xd9
#       jpg     y       20000000        \xff\xd8\xff\xe? \xff\xd9

#       jpg     y       20000000        \xff\xd8       \xff\xd9
#
# PNG   (used in web pages)
#       png     y       200000  \x50\x4e\x47?  \xff\xfc\xfd\xfe
#
#
# BMP   (used by MSWindows, use only if you have reason to think there are
#       BMP files worth digging for. This often kicks back a lot of false
#       positives
#
#       bmp     y       100000  BM??\x00\x00\x00
#
# TIF
#       tif     y       200000000       \x49\x49\x2a\x00
#
#---------------------------------------------------------------------
# ANIMATION FILES
#---------------------------------------------------------------------
#
# AVI (Windows animation and DiVX/MPEG-4 movies)
#       avi     y       4000000 RIFF????AVI
#
# Apple Quicktime
#   Some users have reported that when using these headers that the
#   headers repeat inside the files. This can generate lots of smaller
#   output files. You may want to consider using the -q (quick mode)
#   flag to avoid this problem.
#
#       mov     y       4000000 ????????\x6d\x6f\x6f\x76
#       mov     y       4000000 ????????\x6d\x64\x61\x74
#
# MPEG Video
```

```
#       mpg     y       4000000 mpg     eof
#       mpg     y       20000000 \x00\x00\x01\xba      \x00\x00\x01\xb9
#       mpg      y      20000000 \x00\x00\x01\xb3     \x00\x00\x01\xb7
#
# Macromedia Flash
#       fws     y       4000000 FWS
#
#---------------------------------------------------------------------
# MICROSOFT OFFICE
#---------------------------------------------------------------------
#
# Word documents
#
# look for begin tag and then wait until the next one (NEXT TAG) -- usually word
documents
# and other Ole2 structured storage files are 'near' each other.  Just make the file
# size large enough to catch our maximium size file.  Look in the audit file to see if
any were chopped.
#
#       doc     y       12500000  \xd0\xcf\x11\xe0\xa1\xb1\x1a\xe1\x00\x00
\xd0\xcf\x11\xe0\xa1\xb1\x1a\xe1\x00\x00 NEXT
#       doc     y       12500000  \xd0\xcf\x11\xe0\xa1\xb1
#
# Outlook files
#       pst     y       400000000      \x21\x42\x4e\xa5\x6f\xb5\xa6
#       ost     y       400000000 \x21\x42\x44\x4e
#
# Outlook Express
#       dbx     y       4000000 \xcf\xad\x12\xfe\xc5\xfd\x74\x6f
#       idx     y       4000000 \x4a\x4d\x46\x39
#       mbx     y       4000000 \x4a\x4d\x46\x36
#
#---------------------------------------------------------------------
# WORDPERFECT
#---------------------------------------------------------------------
#
#       wpc     y       100000  ?WPC
#
#---------------------------------------------------------------------
# HTML
#---------------------------------------------------------------------
#
#       htm     n       50000   <html                 </html>
#
#---------------------------------------------------------------------
# ADOBE PDF
#---------------------------------------------------------------------
#
#       pdf     y       5000000 %PDF-  %EOF
#
#
#---------------------------------------------------------------------
# AOL (AMERICA ONLINE)
#---------------------------------------------------------------------
#
# AOL Mailbox
#       mail    y       500000   \x41\x4f\x4c\x56\x4d
#
#
#
#---------------------------------------------------------------------
# PGP (PRETTY GOOD PRIVACY)
#---------------------------------------------------------------------
#
# PGP Disk Files
#       pgd     y       500000  \x50\x47\x50\x64\x4d\x41\x49\x4e\x60\x01
#
# Public Key Ring
#       pgp     y       100000  \x99\x00
# Security Ring
#       pgp     y       100000  \x95\x01
```

```
#       pgp     y       100000  \x95\x00
# Encrypted Data or ASCII armored keys
#       pgp     y       100000  \xa6\x00
# (there should be a trailer for this...)
#       txt     y       100000  -----BEGIN\040PGP
#
#
#---------------------------------------------------------------------
# RPM (Linux package format)
#---------------------------------------------------------------------
#       rpm     y       1000000 \xed\xab
#
#
#---------------------------------------------------------------------
# SOUND FILES
#---------------------------------------------------------------------
#
#       wav     y       200000  RIFF????WAVE
#
# Real Audio Files
#       ra      y       1000000 \x2e\x72\x61\xfd
#       ra      y       1000000 .RMF
#
#---------------------------------------------------------------------
# WINDOWS REGISTRY FILES
#---------------------------------------------------------------------
#
# Windows NT registry
#       dat     y       4000000 regf
# Windows 95 registry
#       dat     y       4000000 CREG
#
#
#---------------------------------------------------------------------
# MISCELLANEOUS
#---------------------------------------------------------------------
#
#       zip     y       10000000        PK\x03\x04      \x3c\xac
#
#       java    y       1000000 \xca\xfe\xba\xbe
#
#---------------------------------------------------------------------
# ScanSoft PaperPort "Max" files
#---------------------------------------------------------------------
#      max    y      1000000    \x56\x69\x47\x46\x6b\x1a\x00\x00\x00\x00
\x00\x00\x05\x80\x00\x00
#---------------------------------------------------------------------
# PINs Password Manager program
#---------------------------------------------------------------------
#      pins   y      8000     \x50\x49\x4e\x53\x20\x34\x2e\x32\x30\x0d
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] RCFL Program Annual Report for Fiscal Year 2003.

[2] Prosise, Chris,and Mandia, Kevin, and Pepe, Matt. *Incident Response and Computer Forensics, Second Edition* McGraw-Hill Osborne Media, 17 July 2003.

[3] IRS Criminal Investigation Electronic Crimes Program. "ILOOK Investigator Toolset". http://www.ilook-forensics.org/. 2005. Last Visited: March 2005.

[4] Access Data. "Forensic Toolkit". http://www.accessdata.com/. 2005. Last Visited: March 2005.

[5] Guidance Software. "Encase". http://www.guidancesoftware.com/. 2005. Last Visited: March 2005.

[6] UNIX Man Pages, "FILE". Last visited: March 2005.

[7] UNIX Man Pages, "STAT(2)". Last visited: March 2005.

[8] Digital Imaging Group, "DIG2000 file format proposal", Appendix A, October 1998.

[9] The Chicago Project: http://chicago.sourceforge.net/, 2002. Last visited: January 2005.

[10] Sun Microsystems. "OpenOffice". http://www.openoffice.org/. 2005 Last Visited: March 2005.

[11] Adobe Systems Incorporated, "Portable Document Format Reference Manual Version 1.3", 11 March 1999.

[12] Kornblum, Jesse and Kendall, Kris. "Foremost 0.69", http://foremost.sourceforge.net/. 2004. Last visted: March 2005.

[13] Hamilton, Eric. *JPEG File Interchange Format, Version 1.02. 1* September 1992

[14] Joint Photographic Experts Group, "JPEG 2000 Specification" http://www.jpeg.org/jpeg2000/, 2004. Last visited: March 2005.

[15] CompuServe Incorporated.,"GRAPHICS INTERCHANGE FORMAT(sm)", July 1990

[16] Wouters, Wim. "BMP Format", February 1997.

[17] Apple Computer, Inc., "QuickTime File Format Specification", May 1996.

[18] Microsoft Corporation, "Advanced Systems Format (AFS) Specification Revision 01.20.02", June 2004.

[19] PKWARE Inc. ".ZIP File Format Specification Version: 6.2.0", June 2004.

[20] Wilson,Scott. "WAVE PCM soundfile format", http://ccrma.stanford.edu/courses/422/projects/WaveFormat/, 2003. Last visited March 2005.

[21] McGowan, John. "AVI Overview", http://camars.kaist.ac.kr/~jaewon/special/avi/avi.html , 1997. Last visited March 2005.

[22] R.S. Boyer, and J.S. Moore, A Fast String Searching Algorithm., *Communications of the Association for Computing Machinery*, 20(10), 1977, pp. 762-772.

[23] Komoncharoensiri, Jamras. "String Searching and Replacement", http://www.4d.com/docs/CMU/CMU79780.HTM, 2001. Last visited December 2004.

[24] Bovet, Daniel, and Cesati, Marco. *Understanding the LINUX KERNEL*. Oreilly, Sebastopol, 2001.

[25] Carrier, Brian. "Digital Forensics Tool Testing Image (#8)", http://dftt.sourceforge.net/test8/, 2004. Last visited January 2005.

[26] Johnsonbaugh, Richard, and Kalin, Martin. *Applications Programming in ANSI C. 3rd Ed*., Prentice Hall, New Jersey, 1996.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chris Eagle
   Naval Postgraduate School
   Monterey, California

4. Dr. George Dinolt
   Naval Postgraduate School
   Monterey, California

5. Cynthia Irvine
   Naval Postgraduate School
   Monterey, California

6. Nick Mikus
   Naval Postgraduate School
   Monterey, California